Conversion of Decision Tables To Computer Programs

LAURENCE I. PRESS
San Fernando Valley State College, Northridge, Calif.

Several translation procedures for the conversion of decision tables to programs are presented and then evaluated in terms of storage requirements, execution time and compile time. The procedures are valuable as hand-coding guides or as algorithms for a compiler. Both limited-entry and extended-entry tables are analyzed. In addition to table analysis, the nature of table-oriented programming languages and features is discussed. It is presumed that the reader is familiar with the nature of decision tables and conventional definitions.

1. Introduction

This paper is concerned with the production of computer programs from decision tables. It is assumed that the reader already has a knowledge of the conventions used in constructing them. For an excellent description of decision tables, their advantages and application areas, the reader is referred to [5].

Decision tables have been shown to be an effective device for the communication of job definitions between persons both within and outside of the data processing area. In addition, they may be an extremely useful aid in communicating with a computer, either via an automatic programming system or as a hand-coding tool.

In hand coding, a decision table is somewhat analogous to a flowchart. However, in ease of converting complex systems of conditional rules to efficient programs, insuring completeness, and guiding program organization, decision tables seem to offer significant advantages. For an example of such a complex program, and hand-coding experience, see [13].

Regardless of the method for translation, orderly procedures (algorithms) are needed in order to achieve efficient programs from decision tables. Possible criteria for the efficiency of such procedures are translation time, storage requirements of the programs produced, and execution time of the programs produced.

Three approaches to the conversion of conventional, limited-entry tables are presented in Section 2 of the paper. They are evaluated in terms of the criteria mentioned above. The next section examines the more generally useful cases of limited-entry tables, where ambiguity is allowed, and that of extended-entry tables, which are shown to be identical for purposes of programming. In the final section decision-table language structures and features are discussed.

All of the techniques presented are applicable either for manual or automatic programming. Formal proofs of optimality are not included (seldom is optimality asserted); however, an attempt has been made to illustrate the reasoning behind each decision or assertion.

2. Limited-entry, Nonambiguous Tables

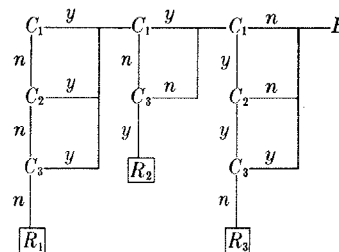
This section deals exclusively with the processing of conventional limited-entry tables. The term *nonambiguous* merely infers the usual restriction against redundancy or contradiction between rules. The concept of ambiguity is discussed in more detail at the beginning of Section 3.

In attempting to devise a procedure for the conversion of limited-entry tables to computer programs, let us consider the example shown in Figure 1.

		Rules 1	2	3	*E*
Conditions	1	*N*	*N*	*Y*	
	2	*N*	*I*	*Y*	
	3	*N*	*Y*	*N*	

FIG. 1

A technique yielding a network that accurately reflects the logic specified in the table is testing the rules one at a time until one is found where all conditions are satisfied. At that point, the actions associated with that rule may be executed. This procedure might result in the following flowchart from Figure 1.



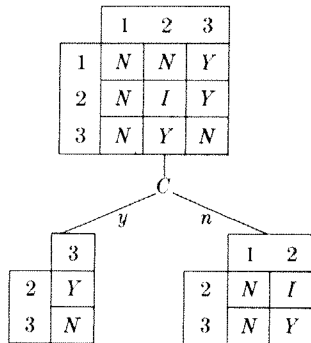
To minimize execution time, the rules would be ordered on relative frequency of occurrence per number of nonindifferent entries prior to examination. It should be noted that the entire network might be transversed in choosing a rule.

In general, this technique would result in a network which has as many branch points as there are nonindifferent entries in the table. A procedure which would result in less branch points would save computer storage.

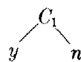
Let us try a different general approach. Rather than examining the rules individually, we may test the various conditions in order, which eliminates rule(s) from further consideration. This philosophy should be more efficient with respect to storage, as a single test can eliminate several rules, whereas with the previous method several tests are often needed to examine a single rule. Furthermore, the outcome of early tests is "forgotten" when looking at other rules.

To illustrate this general approach, consider again Figure 1. If condition 1 is tested and found to be true,

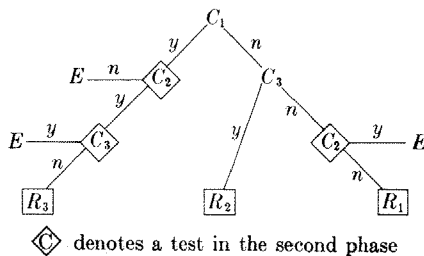
rules 1 and 2 may be eliminated from further consideration. In either case E (error or else) is still possible. In making a test such as just described, we have parsed Figure 1 into two subtables by discriminating on the condition in row 1, the key row. This step may be symbolized as follows:



or more simply:



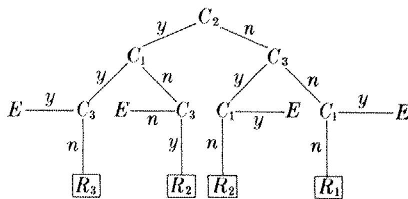
An entire network may be constructed by repeating this process for each subtable produced, until all subtables consist of only one rule (and possibly E). At the point where only one rule has not been eliminated, a second phase must be entered in order to explicitly test all previously untried conditions due to the possibility of an E-condition. Continuing with our example, the following network might result:



◇ denotes a test in the second phase

This network with only five branch points, requires less storage than that depicted previously; however, it was constructed in a rather arbitrary fashion. For example, why test condition 1 first rather than 2 or 3?

We must devise an orderly procedure for choosing the test condition which should be used in parsing a given table in order to yield a network with a minimum number of branch points. If it is not clear that an arbitrary choice will not suffice, consider the network which might have resulted from Figure 1 if C_2 had been first:



It is not surprising that this network is less efficient (seven branch points) than the first, because a certain rule (2) was indifferent to the state of the first condition tested and hence could not be eliminated from further consideration regardless of the result of the test of C_2.

In general, if we parse a table on a given condition (C_k) and

n = number of rules in the table,
m = number of conditions in the table,
y = number of Y entries in C_k,
x = number of N entries in C_k,
i = number of I entries in C_k,

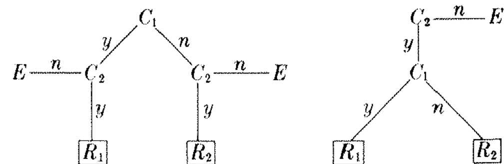
two subtables result, each having $m-1$ conditions and $x+i$ and $y+i$ rules, respectively. Clearly, if we can always discriminate on a condition which minimizes the value of i, smaller subtables, and hence more efficient networks, result. Therefore, the first step in our procedure for choosing the key condition is as follows:

Step 1. Choose the key condition from the set of rows with the minimum number of indifferent entries.

Perhaps an arbitrary choice from this set would suffice; however, in considering the following example we see that this is not the case:

	1	2
1	Y	N
2	Y	Y

Both C_1 and C_2 satisfy rule 1, as neither contains any indifferent entries; however, either of the following networks may be derived if only step 1 is followed.



The network on the right requires less storage. It is obvious that by testing a condition with all Y or N entries, we may save branch points in subsequent second phase portions of the network. In a larger table this saving clearly is even greater. Therefore, our second step might be:

Step 2. If one of the rows contains all Y or all N entries, discriminate on that condition.

We now have two steps which lead to efficient choices of key rows. It should be noted that these may be applied by looking at the conditions individually. In order to derive a more efficient network, it is necessary to take cognizance of the relationships between the various conditions as well as the nature of the conditions when considered individually. It is in overlooking this necessity that many of the "optimal" procedures described in the literature fail.

In formulating a third step, we want to parse the table in a manner which will enhance the possibility of being

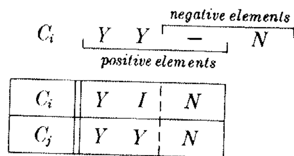
able to apply rule 2. Our goal is to parse the table so that future subtables will contain rows of all Y or all N. Let us consider this table:

	1	2	3
1	Y	Y	N
2	Y	N	Y
3	N	N	Y

Applications of rules 1 and 2 imply indifference between the various rows for the first discrimination. However, if condition 1 or 3 is chosen first, it is apparent that the resultant subtable will contain rows of all N and all Y, respectively. On the other hand, an initial choice of C_2 does not lead to such a fortuitous state and indeed the resultant network requires 6 branch points as compared to 5 if C_1 or C_3 is used first. In order to state a rule which will allow us to employ a "look ahead" strategy such as this, some terms must be defined.

A condition row C_i is made up of two sets of elements, those with Y or I (the positive elements) and those with N or I entries (the negative elements). C_i is said to be complemented by C_j if the negative rules in C_i have only Y or only N elements in C_j and/or the positive rules in C_j have only Y or only N elements in C_j. Furthermore, the number of complementary rules will be known as the count of C_i with respect to C_j (C_{ij}).

As an example, consider the following illustrations:



As the positive elements in C_i have only Y elements in the corresponding rules of C_j, C_j complements C_i. As this match occurs for two rules, $C_{ij} = 2$.

We may now state a third step which may be taken after the first two lead to seeming indifference between several rows.

Step 3. Discriminate on the condition row which maximizes CS_i where CS_i equals $\sum_{j=1}^{n} C_{ij} - C_{ii}$ and where n equals the number of conditions in the table.

Translation of the following table illustrates the application of the steps:

	1	2	3	4
1	Y	Y	Y	Y
2	Y	N	Y	N
3	Y	I	N	Y
4	N	Y	N	N
5	Y	Y	Y	N

Step 2 clearly indicates that condition 1 should be used for the entire table.

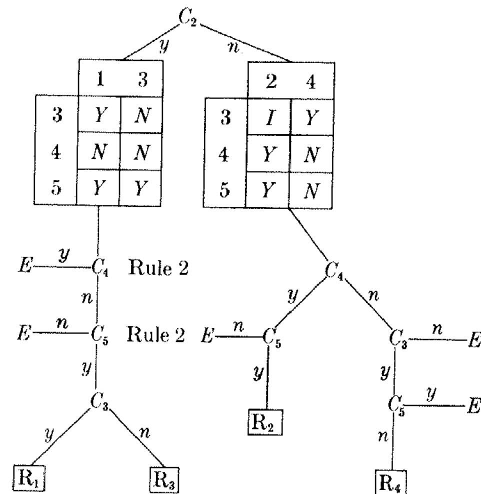
C_1 —n— E
y

	1	2	3	4
2	Y	N	Y	N
3	Y	I	N	Y
4	N	Y	N	N
5	Y	Y	Y	N

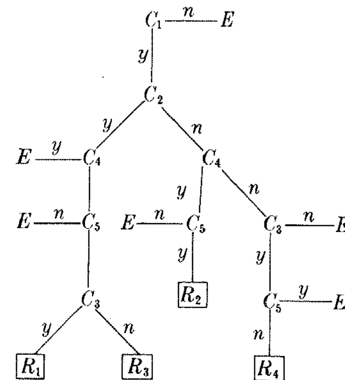
Step 1 tells us that the key row should be 2, 4 or 5. Furthermore:

$$CS_2 = C_{2,3} + C_{2,4} + C_{2,5} = 0 + 2 + 2 = 4$$
$$CS_4 = C_{4,2} + C_{4,3} + C_{4,5} = 1 + 0 + 1 = 2$$
$$CS_5 = C_{5,2} + C_{5,3} + C_{5,4} = 1 + 1 + 1 = 3.$$

Step 3 dictates that C_2 be used for the next discrimination.



Showing the complete network, we have:



Thus far, we have stated two procedures for deriving programs from decision tables. The number of branch points in a network derived via the successive parsing procedure is less than or equal to the number required when the sequential examination of rules is performed. (It is equal only in the cases of a 1-rule table or a 2-rule table with only a single nondifferent entry in one of the rules.) However, except in the case noted, the successive parsing technique requires a greater number of branch points than the number of conditions in the table. This latter fact leads us to examine a third technique which might yield better results with respect to storage requirements.

Execution of this third technique proceeds in two phases.

1. Test each condition and build a mask word reflecting satisfaction (1, 0) or failure (0, 1) of each in a 2-bit entry.

2. Match this mask word against a set of words which are derived from the rules where a 1, 0 denotes a Y element, 0, 1 an N element and 1, 1 an I element.

The actions associated with the first rule found to match the condition-state word will be performed.

This procedure, as has been noted previously, minimizes the number of branch points in the network. However, this is not as valid a measure of storage usage as it is in the previous cases. It is clear that the mask words associated with each rule must be present at object time, as well as the routine for matching them. It is necessary to evaluate these additional storage requirements further, to determine their importance as compared to additional branch points.

Using the IBM System/360 as a vehicle, the absolute minimum marginal storage requirement imposed by a branch point is 4 bytes, i.e.:

$$CR \quad A, B$$

$$BC \quad X'8'$$

Even this sequence displaces enough storage to accommodate the entries of a 4×4 table. When one considers the evaluation of expressions, rather than simple operands which are assumed to be in registers, it is clear that the storage required for a single branch point could easily be sufficient to accommodate the masks necessary for a large table. The storage required for the interpretive routine must be considered also; however, a single, common subroutine will be shared by all tables in a program.

This technique seems to offer an advantage in terms of storage usage; however, the dimension of execution time must also be considered. The first phase, it will be recalled, requires testing *all* of the conditions in the table—this would be the worst possible case were the parsing approach used. For a presentation of an algorithm which, given relative frequency of success for each rule, purports to develop a parsing network of optimal execution time, the reader is referred to [15].

Execution time is also adversely affected by the matching routine. Ordering the rules on expected relative frequency of success in some part enhances execution times.

The final consideration in evaluating these procedures is the time required for translation from table to program. There can be no question that building rule masks and a serial set of condition tests is easier than applying the parsing procedure which has been described, whether by a human programmer or an automatic translator. Processing into a matching program would seem to be slightly simpler than applying the first approach described and would require less time due to the fact that a smaller program would result.

Summary. The various approaches may be summarized as follows:

4	*Varies*	2
2	2	3
3	1	3
1	3	1

3. Ambiguous and Extended-entry Tables

Thus far, the discussion has assumed only limited-entry tables without ambiguity. A table is said to be *ambiguous* if it is logically possible that a set of condition states occur such that more than one rule be satisfied. It is important to note that a table may be ambiguous even though the nature of the conditions being tested implicitly resolve the problem. For example,

C_1	I	Y
C_2	Y	I

is ambiguous even though the systems analyst might know that C_1 and C_2 cannot simultaneously be satisfied. An ambiguous table is recognized to be one that contains either redundant or contradictory rules. For a more rigorous discussion, see [12].

At this point, one might reasonably question the necessity for being concerned with ambiguous tables. As a matter of fact, these are of paramount importance due to:

(1) the common case where the systems analyst is aware of implicit information in the problem, and is burdened by making it explicit in order to eliminate ambiguity;

(2) the fact that extended-entry tables are essentially equivalent to ambiguous, limited-entry tables, and may easily be mapped into that form.

For handling extended-entry tables, we utilize the technique of reducing the problem to one which has been solved previously. An extended-entry condition takes on one of the following formats:

Stub	*Entry*
1. operand_0 operator	$\text{operand}_1, \cdots \text{operand}_n$
2. operand_0	operator $\text{operand}_1, \cdots,$ operator operand_n
3. operand_0 vs. operand_1	$\text{operator}_1, \cdots, \text{operator}_n$

These three formats may be transformed as follows:

1. Create a separate condition for each entry operand by combining operand_0, the operator, and the appropriate entry operand in the stub portion, and by entering a Y in the corresponding rule. All other rules are indifferent with respect to this condition.

2. Same as for 1, but bring the entire operator/operand pair to the stub portion.

3. Same as for 1, but bring the operator into the stub.

To illustrate, we transform the table:

A	$=B$	$>C$	$\neq B$
M,N	$\geq$	$<$	--
$X<$	Y	--	W

into:

$A = B$	Y		
$A > C$		Y	
$A \neq B$			Y
$M \geq N$	Y		
$M < N$		Y	
$X < Y$	Y		
$X < Y$			Y

Notice that we have transformed the extended-entry table into an ambiguous, limited-entry table. Making use of the definitions of the operators and possible equivalence of the operands, we may be able to condense the resultant table in a second phase. Doing so, our example simplifies to:

$A = B$	Y		N
$A > C$		Y	
$M \geq N$	Y	N	
$X < Y$	Y		
$X < W$			Y

We may now consider the effect of ambiguity on the procedures described in the preceding section. In the case of a parsing technique, a subtable evolves which contains only one condition and is of a form other than:

Y	N

or

N	Y

While this technique readily recognizes ambiguity, it is unable to cope with it.

The other procedures described offer no such simple test for recognizing ambiguity; however, they are able to successfully process ambiguous tables. This capability results from the fact that the rules are examined independently and the first one which is satisfied is accepted; hence the ambiguous situation is resolved.

A two-phase translation to ambiguous, limited-entry form enables extended-entry tables to be processed by the same procedure as limited-entry tables for the preparation of programs. Due to the ability to resolve ambiguity, simplicity and speed of translation, and storage economy, the mask-matching technique seems very desirable except in the case where execution time is critical and one of the rules in the table is highly dominant in terms or relative frequency of success per number of nonindifferent elements.

4. Decision Table Languages

Most activity in the area of languages to date has been in translating tables into some intermediate language (FORTRAN, COBOL, etc.) and allowing the standard compilers to carry the process of machine language. This philosophy has prevailed as all work to date has been informative and experimental, and therefore the most expedient implementation of a processor was sufficient.

Justification for an independent decision-table language could spring from any of several sources such as an increase in object time efficiency, reduced compile time or most important, the need for language features or structure which do not lend themselves to implementation through a secondary language. Object time efficiency depends primarily upon the intermediate language processor, and it would seem that it could be controlled at that level. The costs of two-stage translation may be important, but are diminished by trends toward monitored operations and the residence of programming systems on nonsequential devices. Finally, work to date has not produced definitions of language features or of a general language philosophy which cannot be implemented via any of several current programming languages.

The above seems to indicate that no compelling reason exists to implement a decision-table processor which is independent of an intermediate language. The choice of the intermediate language influences many facets of the decision table language such as variable-naming conventions, statement referencing, allowable action formats, etc.; however, the general nature of the decision-table language is relatively insensitive to this choice.

Implementation via an intermediate language does raise one major question as to the philosophy or nature of the decision-table language, that of table/intermediate language dominance. It is possible to formulate a useful language where the programmer works entirely in terms of decision tables. He is isolated from the intermediate language per se to the greatest extent possible, and, when forced to use certain statements, he may regard them as isolated features of the decision-table language. The alternative approach is to view the ability to process decision tables as an adjunct of the intermediate language, with the ability to freely utilize the features of both forms in solving a problem. In this latter case, neither the intermediate language nor decision tables are dominant, but they are at the same level.

The table-dominant approach would seem to offer advantages in terms of ease of learning, complete problem orientation, etc. However, is this a truly valid observation? Would it not be possible to present a well-chosen subset of a nondominant language and obtain the same simplifying advantages?

More important, it must be recognized that relatively few computer jobs lend themselves to formulation entirely in terms of decision tables. It may be possible to broaden the applicability of a table-dominant language by forcing the formulation of certain other jobs to conform to decision-table conventions via awkward and unusual use of the language (e.g. unconditional tables). On the other hand, a very large class of jobs exist wherein certain portions are well suited to formulation as decision tables (flow of control, logic, data validity, etc.), and other portions best stated as decision tables (assuming that it is possible to do so).

It is in the latter area that primary justification for a processor exists and where a nondominant approach is

dictated. It would seem that the table-dominant approach greatly restricts the range of usefulness of a decision table language by "hiding" capabilities and presents no compelling advantages in terms of language power or processor implementation ease.

In addition to considerations such as those above, several table-oriented features should be included in a decision table language. Following is a brief description of several of the more important facilities, many of which have been postulated or implemented elsewhere.

1. The programmer should have the option of ranking storage utilization, execution time and translation time in order of preference.

2. The programmer should have the option of causing the second phase of a parsing procedure to be deleted, i.e., accept a rule after all, and then one rule has been eliminated. This implies no error-else actions and results in economies in every aspect.

3. Statistical analysis aids should be included in the language. For example, the programmer should be able to cause a count to be kept of the frequency of acceptance of each rule. These figures are necessary as input to the compiler if execution time is to be minimized, and they could be revised periodically via such a tally feature. In addition, statistics on the states of individual conditions might be of interest to the systems analyst.

4. Another class of statistics which would be of use to the systems programmer is concerned with the frequency of utilization of various language features and options. This type of information is seldom available in representative form and great effort is spent to estimate it. With the advent of monitored operation, this information is easily gathered and maintained and is clearly of use with regard to all programming systems, not merely to decision table languages.

RECEIVED OCTOBER, 1964; REVISED FEBRUARY, 1965

REFERENCES

1. ARMERDING, G. W. FORTAB: a decision table language for scientific computing applications. Mem. RM-3306-PR, Rand Corp., Santa Monica, Sept. 1962.
2. CANTRELL, H. N., KING, J., AND KING, F. E. H. Logic structure tables. *Comm. ACM 4* (June 1961), 272–275.
3. CODASYL Systems Group. DETAB-X, preliminary specifications for a decision table structured language. Sept. 1962.
4. CODASYL Systems Group. Decision table tutorial using DETAB-X. Rev. ed., Oct. 1962.
5. DIXON, P. Decision tables and their application. *Comput. Automat.*, (Apr. 1964).
6. EGLER, J. F. A procedure for converting logic table conditions into an efficient sequence for test instructions. *Comm. ACM 6* (Sept. 1963), 510–514.
7. MONTALBANO, M. Egler's procedure refuted. (Letter to the Editor) *Comm. ACM 7* (Jan. 1964), 1.
8. EVANS, O. Y. Advanced analysis method for integrated electronic data processing. IBM Gen. Inform. Man. #F20-8047.
9. GLANS, T. B., AND GRAD, B. Tabular descriptive language. IBM Tech. Rep. No. 2A5, Jan. 1962.
10. GRAD, B. Tabular form in decision logic. *Datamation* (July 1961).
11. KAVANAGH, T. F. TABSOL—a fundamental concept for systems design. Proc. 1960 Eastern Joint Comput. Conf.
12. MONTALBANO, M. Tables, flow charts, and program logic. *IBM Syst. J.* (Sept. 1962).
13. NICKERSON, R. C. An engineering application of logic—structure tables. *Comm. ACM 4* (Nov. 1961), 516–520.
14. POLLACK, S. L. Analysis of decision rules in decision tables. Mem. RM-3669-PR, Rand Corp., Santa Monica, May 1963.
15. ——. Conversion of limited-entry decision tables to computer programs. Mem. RM-4020-PR, Rand Corp., Santa Monica, May 1964.

Do You Know That COMPUTING REVIEWS KWIC INDEX Lists

293 SEPARATE ARTICLES

on

BUSINESS AND MANAGEMENT DATA PROCESSING?

These are referenced under the Key

MGNTDP

in one continuous list on pages 219–220.

All aspects of Management and Business are covered:

Manufacturing • Marketing • Merchandising
Financial • Distribution • Transportation • Communication
Research

CR KWIC INDEX IS AVAILABLE FROM

Association for Computing Machinery, 211 East 43 Street, New York, N. Y. 10017, $15.00