H. J. Myers

Compiling Optimized Code from Decision Tables

Abstract: This paper reviews the structure of decision tables and methods for converting them into procedural code. It describes new optimization methods, which are applied before, during, and after code generation. Some results from an experimental decision table processor are provided.

Introduction

Decision tables have been in use for over ten years, principally in business applications, to state problems that contain a relatively high proportion of programmed tests. Numerous compilers have been built that convert the logic expressed in decision tables into algorithms that are executable by computer. Emphasis in decision-table compilers has typically been placed on producing logically correct code, on checking the decision table for completeness and consistency, and on ordering condition tests for efficient execution. Most decision table compilers produce code that is in a higher-level language, leaving optimization of the produced object code up to the high-level language compiler. However, the structure of the code produced by typical decision-table compilers is of a type that is improved little by any of the optimizing algorithms used by commercial compilers today.

General-purpose support programs (systems programs) are also typified by a high proportion of programmed tests. It would therefore appear that effective use of decision tables could be made in describing systems programs. However, systems programs must also be comprised of highly optimal code. This paper describes some decision-table compiling algorithms that provide a program structure of high enough quality to satisfy most systems programming needs. The output can be used either by a post-processing compiler or by a programmer as a guide in hand coding.

For the project, we constructed a running compiler and support system into which were introduced numerous decision tables based on actual systems programs. From these decision tables, procedural code (in PL/I format) was produced. Some samples of the code structure produced are included in the Appendix to enable the reader to judge the effectiveness of the compiler. The system was produced in an interactive APL environment, which allows considerable flexibility in revising and augmenting algorithms.

In the paper, we first review the structure of decision tables and the procedures used to map them into code. However, the main emphasis is on the optimization methods we use before, during, and after code generation.

Decision tables

In order to make this paper reasonably self-contained, we review here the structure of decision tables and the general methods used to convert them into procedural language. We attempt to emphasize those aspects of the process that provide opportunities for optimization.

A sample decision table is shown in Fig. 1.

The *stub* portion (2) of the table gives descriptions of *conditions* (5,6) *actions* (7,8) and *exits* (9,10). The format of the stub contents is generally constrained by the target language into which the table is being translated. Our processor places no constraint on the contents of the stubs; if correct PL/I code is to be produced, the condition stubs must contain one PL/I relational expression each; the actions should contain a PL/I assignment statement, call statement (or other nonbranching execut-

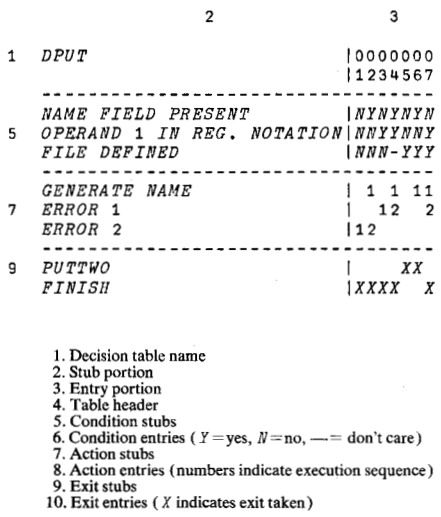


Figure 1 Sample decision table.

able statement); and the exit stubs should contain only a valid PL/I name. The table name should also be valid in PL/I.

Each column in the *entry* portion (3) of the table represents a *rule*. Rule numbers (two-digit numbers read vertically) and the table name appear in the *table header* (4). Rule 02 in Fig. 1, for example, indicates that if a *NAME FIELD IS PRESENT* (*Y* means yes) *and* if *OPERAND 1 IS NOT IN REGISTER NOTATION* (*N* means no) *and* if *FILE IS NOT DEFINED*, then the actions taken are *GENERATE NAME* and *ERROR 2* (in that order), followed by an exit to *FINISH*. In general, a rule specifies that for a unique combination of conditions, some selected actions are performed and a selected exit is taken.

The decision-table representation of logic does not impose any strict ordering on the sequencing of condition tests. Furthermore, actions in a given table are not allowed to modify factors that would cause a change in the outcome of a condition test in the same table. This gives the compiler more latitude in selecting an optimal ordering of condition tests with respect to each other and with respect to actions.

The ordering of actions with respect to each other can be loosely defined. In Fig. 1 the order required is specified by an integer opposite a selected action. If there is no integer, the action is not selected. Within a given rule, actions are executed in the order specified (e.g., in rule 07 *GENERATE NAME* occurs before *ERROR 1*). Exactly one exit is taken after execution of the actions of the selected rule.

In this experiment, we consider limited-entry type decision tables, in which the value of a condition is limited to yes or no. The alternative, extended entry tables, allows values that are numbers or number ranges. However, as Press [1] demonstrates, these can be converted into limited-entry tables, so that our methods apply to both types of decision tables.

The occurrence of a third value (–) in rule 04 of Fig. 1 does not contradict the two-value restriction. Rule 04 is actually a condensation of two rules (say 4a and 4b), which have identical action and exit entries, but whose condition entries are:

Condition	*4a*	*4b*	*4c*
1	*Y*	*Y*	*Y*
2	*Y*	*Y*	*Y*
3	*Y*	*N*	-

Rules 4a and 4b mean that the specified actions and exits are to be performed when conditions 1 and 2 are both yes, regardless of the value of condition 3. Therefore in rule 04, condition 3 is immaterial—a don't-care condition. As will be shown later, in the code produced by the compiler, there will be no test for condition 3 in the flow path for which conditions 1 and 2 are both true.

Because conditions in a limited-entry table are binary valued, there are 2^n unique combinations of the values of n conditions, and therefore 2^n rule columns. In practice, the number of rules is greatly reduced by the introduction of don't-care entries and consolidation of the rules as shown for rule 04. If rules 4a and 4b had been counted as part of the table in Fig. 1, that table would have 8 (2^3) rules for 3 conditions. A detailed discussion of column combining is presented later in the section on pregeneration optimization.

With the 2^n requirement on the rule count, one can program a verification of the rule entries that checks for both consistency and completeness. However, as will be seen, the check does not extend to an interpretation of the stub information and so it cannot be regarded as a complete check.

Decision-table code production

One of three methods is generally used to map decision table logic into procedural representation. In the *rule mask* method [2–4] every condition is first tested and then a *selection mask* is created. Next, for each action and exit, an *action mask* (created at compile time) is compared to the selection mask. If the masks coincide, the action (or exit) is performed.

In the second method, a variation on the rule-mask method, a unique power of 2 is assigned to each condition, which is then tested and its number added to a counter if the condition is true. At the end of condition testing, the counter is used as an index into a branch table and control is transferred to the appropriate action-exit sequence. (Note that the branch table must have 2^n

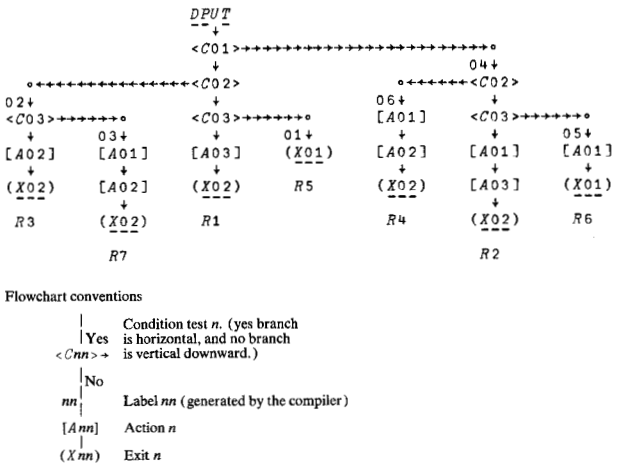


Figure 2 Tree corresponding to sample decision table.

entries for n conditions.) These two methods require that all tests be performed regardless of the don't-care entries in the table. They tend to produce (when bit masks are used) a small program that runs longer than is usually necessary.

We use the third scheme, called the *condition tree* method, which causes the generation of a tree-structured program with condition tests at each node. Each action-exit sequence is placed at the leaves. Figure 2 shows an unoptimized tree corresponding to the decision table of Fig. 1. The rule numbers for each action-exit sequence are placed under the leaves for clarity.

Notice the one-to-one correspondence between the leaves of the tree in Fig. 2 and the rule columns of the decision table shown in Fig. 1. Notice also the large amount of redundancy in the code generated. In this case, the penalty for this redundancy is not lengthened execution sequences, but excessive storage consumption. Figure 3 shows the effect of the optimization algorithms on the tree of Fig. 2.

Figure 4 shows an *abstract* listing of the program diagrammed in Fig. 3. This listing is called abstract because nothing in it refers to the actual (concrete) conditions, actions, or exits that are described in the table stubs. In fact, the processing algorithms ignore the contents of the stubs except for listing purposes.

Because we are principally interested in compilation, the abstract format is the one used in this paper. However, with little difficulty, concrete listings can be produced for any procedural language.

Figure 5 is a concrete listing in PL/I format of the program shown in Fig. 4. Because the stubs in the original table (Fig. 1) do not follow PL/I conventions, the result is *not* a PL/I procedure.

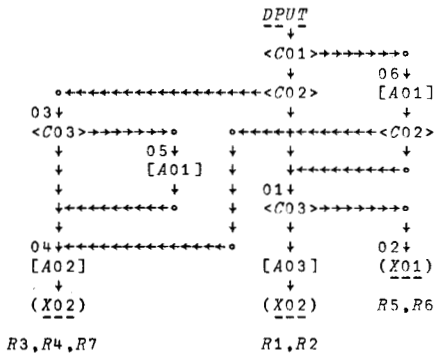


Figure 3 Optimized tree.

Figure 4 Optimized listing.

```
LINE                              BRANCH
NUMBERS LABELS OPERATIONS OPERANDS
000|           C01→06
001|           C02→03
002|01:        C03→02
003|           A03
004|           X02
005|02:        X01
006|03:        C03→05
007|04:        A02
008|           X02
009|05:        A01
010|           ---→04
011|06:        A01
012|           C02→04
013|           ---→01
```

Note: *Cnn*→ indicates a conditional branch, and
---→ indicates an unconditional branch.

Figure 5 Concrete listing.

```
DPUT:   BEGIN;
        IF NAME FIELD PRESENT THEN GO TO DPUT6;
        IF OPERAND 1 IN REG. NOTATION THEN GO TO DPUT3;
DPUT1:  IF FILE DEFINED THEN GO TO PUTTWO;
        ERROR 2;
        GO TO FINISH;
DPUT3:  IF FILE DEFINED THEN GO TO DPUT5;
DPUT4:  ERROR 1;
        GO TO FINISH;
DPUT5:  GENERATE NAME;
        GO TO DPUT4;
DPUT6:  GENERATE NAME;
        IF OPERAND 1 IN REG. NOTATION THEN GO TO DPUT4;
        GO TO DPUT1;
        END;
```

• *Code generation*

Various methods exist for the generation of tree-form code from decision tables [5–7]. Briefly, one selects (by criteria described later) a condition. The rule columns are then partitioned into two groups—a no-group and a yes-group—according to the values in each column of the selected condition. If the condition value is don't-care in a particular column, that column is placed in both

RULE	1	2	3	4	5	6
A1	1	1		1	1*	
A2	1		1*	1		1*
A3	1	1*	1*		1*	
X1	1	1*	1*	1*	1*	1*

Figure 6 Cross-linked action-exit sequences.

groups. Code is then produced for the selected condition. Assume that the line

Cnn → label—*m*

is generated, where *label-m* is a created label. The code generator is then re-entered recursively with a subtable consisting of the no-group of rule columns, but with the selected condition row removed from them. Next, the generated label (*label-m*) is produced and the generator is re-entered with the yes-group columns (without the selected condition row). The resulting structure appears as follows:

Cnn → label—*m*
no—group
Label—*m:* yes—group

Because every action sequence terminates in an exit, control will not flow from the no-group into the yes-group.

It can be seen that the yes- and no-groups are composed of substructures analogous to the main structure. Recursion is terminated when there is no condition remaining to be selected. At this time, there should be exactly one rule column remaining. The action-exit sequence is then produced from that column. Because of the presence of don't-care conditions, a special check must be made before generating each conditional branch. This consists of locating any column in which all of the conditions have don't-care entries. If such a column is found, its action-exit sequence is generated and recursion terminates.

Other special tests are made for handling *else* rules and optimization procedures, which will be discussed later.

An extension of this algorithm, called *cross-linking*, is used in the Preprocessor for Encoded Tables (P.E.T.) processor [8] to reduce the number of instances of actions generated. Each action-exit sequence is compared to the action-exit sequences to its left. If some action and its selected successors exactly match the corresponding elements in some rule to the left, they are linked leftward. For example, in Fig. 6, the cross-linked action-exits are marked by an asterisk. When code is generated, the first linked action is replaced by a branch to the action corresponding to it in the rule it is linked to. This reduces redundancy in the generated actions, providing better code than other processors [8,9].

Most published accounts concerned with object code optimization concentrate on the order of condition testing. Reinwald and Soland [10,14] give algorithms (supported by proofs) for generation of optimal object code with respect to time and space. These algorithms require that storage and time costs be associated with each condition test, but do not take into account the optimization of action sequences.

The P.E.T. processor reduces redundancy in action sequences (a space optimization) but does not remove redundant testing sequences. (It relies upon the table-writer to order condition tests and to introduce don't-cares to provide adequate optimization of testing.) P.E.T. also allows the table writer to place actions into the condition portion of the table. This is called *preconditioning*, and is allowed when an action is to be performed regardless of the outcome of the conditional testing that it precedes.

In summary, previous processors have placed the optimization burden on the user to a large extent. This partially reduces the principal benefit of the decision table—the separation of the logic required from the procedural mechanism that implements it.

A recent paper by Dailey [11] has been published since this work was concluded. Although his approach is different, his optimizations seem to overlap some of those presented here. A careful comparison of his methods to ours has not yet been carried out.

The methods described in this paper attempt to optimize with a minimum amount of guidance from the user. Optimization processes are applied before, during, and after code generation, with major emphasis on the final optimization process. The basic generation process produces excellent code from the point of view of time optimization. It generates no unnecessary tests in a given test sequence. It could be further improved if frequency information were provided (see [12] for the method). Most of our optimization methods are aimed at reducing space requirements. Our experience shows considerable success in removing redundant code, and the program structure is suitable for systems programs.

Optimization techniques

Our pregeneration optimization consists of consolidating rule columns and introducing don't-care values into the rule portion of the table. The optimization methods that are carried out during code generation are concerned with the ordering of the condition tests and the "factoring" of actions. (The latter technique is called *preconditioning* in [8], but is more commonly known in compiler-writing circles as "hoisting.") Postgeneration optimization procedures include removal of duplicate sequences of code and consolidation of duplicate flow paths. These procedures often leave dead code, branch

chains, and other dross in their wake, which is cleaned up by some *scavenger* optimization methods. A final optimization process was introduced into the program that we use to create a concrete PL/I listing. This process is an example of a scavenger procedure. Its purpose is to remove unneeded conditional branches to exits, which are themselves branches. These optimization methods are discussed in the succeeding sections.

• *Pregeneration optimization*

The first optimization method used is that of consolidating rules where possible so as to introduce don't-cares into the condition entries. We call this the *merge* process. Figure 7 shows the initial state of a decision table, which we will use to illustrate the process.

Two rule columns can be combined into one if:

- They are identical except for one condition entry, and
- In the differing entry, one column has *Y* and the other has *N* (neither is a don't-care).

The algorithm used first groups rule columns according to action-exit sequences. In Fig. 7 the grouping gives

```
(07 08 09) (10 11 12 12 14 15 16)
```

(Note that groups containing only one column are ignored.) It then forms subgroups according to don't-care patterns. If two rule columns were

```
YY
NY
--
YN
```

they would have the same don't-care pattern and would be put into the same subgroup. Because there are no don't-cares in the table in Fig. 7, all elements of each group have the same don't-care pattern, so that the groups default to subgroups. Within each subgroup, the condition entries are compared to determine if a difference exists in exactly one position. Two columns from Fig. 7 that qualify can be combined as follows:

```
11              11
45              45
YY              Y
NN              N
YY              Y
NY (DIFFERENT) - (DON'T CARE)
```

The value at the position of difference is replaced by a don't-care value and one of the rules is discarded. Once a pair of rules is consolidated, it is removed from the subgroup because its don't-care pattern has changed. The remaining columns in the subgroup are processed similarly until no further combinations can be found.

```
PUT1|0000000001111111
    |1234567890123456
--------------------
C1  |NYNYYYNYNNNNNYYY
C2  |NNYYYNNYYNNYYNNY
C3  |NNNNYNNNNYYYYYYY
C4  |NNNNNYYYYNYNYNYY
--------------------
A1  |          1111111
A2  | 1 11
A3  |1
A4  |  111
A5  |     111
A6  |    1111
A7  |111111111
A8  |    1
--------------------
X1  |XXXXXXXXXXXXXXXX
```

Figure 7 Decision table initial state.

After all subgroups have been processed, the combined columns are regrouped and reprocessed. This iterative process continues until no new combinations are possible. (Note that the particular combinations that occur depend upon the order in which the columns are matched.)

The effect of this optimization on basic code generation (in the absence of other optimization methods) is to remove unnecessary condition testing. It therefore improves both the space and time costs of the resulting code.

Figure 8 shows the results of the merge process and examples of its effects, as well as the effects of different ordering (in the absence of other optimization methods). On the other hand, in the presence of other optimization procedures, the effect is reduced because of overlap of optimization function. This pregeneration optimization gives the user a better insight into his decision table. It also reduces, at an early stage, the amount of work that the other optimizers have to do. This method generally results in a net reduction in compile time.

Both of the compilers that we examined [8,9] require that the user perform this optimization in order to produce better code. It is often convenient for the user to do part of this optimization himself when the situations giving rise to don't-cares are patently obvious. (It allows him to greatly reduce the number of rule columns he has to deal with.) The use of don't-cares also can be a trick that allows conditional test dependencies (a violation of decision-table ground rules); therefore the implemented merge function was designed to leave original don't-cares intact.

It is instructive to compare our merge process to the electrical engineering problem of circuit simplification. Recall that we identified two groups from Fig. 7 having the same action-exit sequence. Consider rewriting the condition entries of the second of these groups as follows so that they look like a Boolean expression.

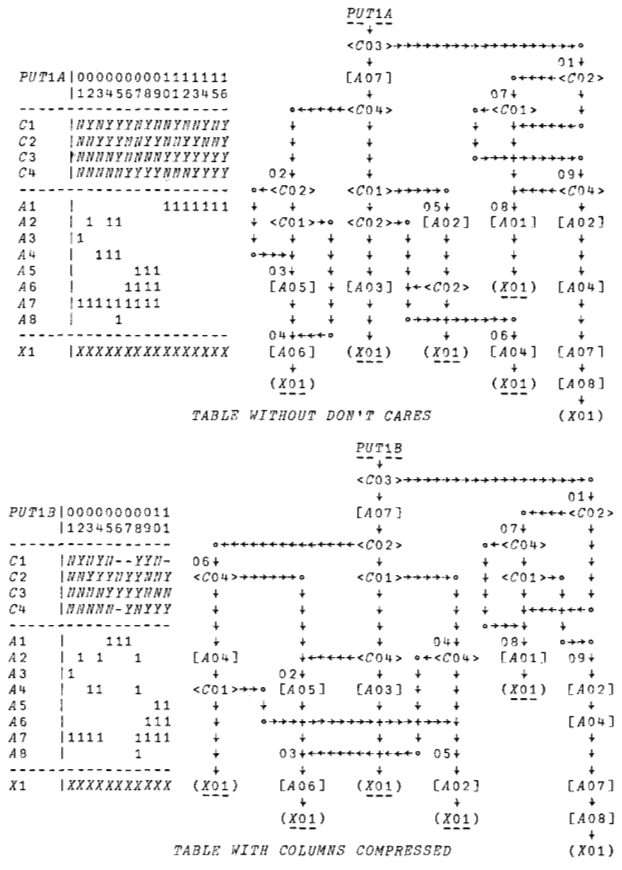


Figure 8 Effects of merge process.

1111111
0123456
NNNNYYY
NNYYNNY
YYYYYYY
NYNYNYY

The equivalent Boolean expression with A, B, C, and D representing conditions 1 through 4, respectively, is:

$$\bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}CD + \bar{A}BC\bar{D} + \bar{A}BCD + A\bar{B}C\bar{D} + A\bar{B}CD + ABCD.$$

Note that each rule represents a term in the expression, and that all rules are in effect ORed (+) together. The elements within a rule column are ANDed. The decision table effectively states in this case, "If *all* of the conditions of *any* column are met, the common action/exit sequence is performed."

Now if we use the Quine-McCluskey technique [13], to find the optimal Boolean expression, the result is:

$$\bar{A}C + \bar{B}C + CD.$$

The equivalent decision-table rule columns are:

N--
-N-
YYY
--Y

Investigation of the code generated from a table so optimized shows no improvement over that for the merge method. This is because the Quine-McCluskey method introduces don't-cares that ultimately cause the code generator to place a rule column in two subtables. It can therefore be concluded that Quine-McCluskey optimization methods are not applicable to the decision-table optimization process. Note that the first two columns actually overlap (e.g., the condition sequence *NNYY* could apply to either one of them). Indeed, if it were not for the fact that both columns specify the same action-exit sequence, the decision table would be inconsistent.

• *Optimization during code generation*

The code generation process we used has already been outlined. However, the method of selecting the "next test" was deferred to this section since it has an impact on the quality of the code produced.

The most significant work published relating to code optimization is, in the author's opinion, that of Reinwald and Soland in Refs. [10] and [14]. Primary papers have also been published by Montalbano [6] and Pollack [7], and contributions made by King [2] and Press [1]. Most recently, Shwayder [12] extends the work of Pollack. King describes the rule-mask technique, which produces compact code but requires that all condition tests be performed regardless of the logic needed. The other papers deal with optimizing methods for sequential testing procedures (the condition-tree method).

Press takes advantage of an *else* column, reducing the number of instances of tests to a minimum (both statically—presence in storage—and dynamically—presence in a flow path). However, the Reinwald and Soland work is the most general and requires that the time and space costs of performing tests be included as input to the optimization process. Reference [10] describes time optimization and Ref. [14] describes space optimization. Both papers provide formulas for calculating the extra cost involved in performing test i after test j has been performed. They next demonstrate a means for determining a lower bound on the extra costs of all tests performed after test j. Then they provide an algorithm for searching a subset of all possible generated testing sequences to find the sequence with minimal lower bound. They prove this sequential procedure to be optimal *in terms of the number of tests*. The costs of actions were not considered. This, we presume, is because action

sequences were considered by them to be atomic units and therefore could not be subjected to reorganization. It should be noted that the time consumed by the search algorithm goes up rapidly with the number of condition tests in the table. (This approach falls in the area of combinatorial mathematics and is akin to the "traveling salesman" problem.)

Our initial approach to test selection followed along the lines suggested by Pollack [7]. It was the simplest, and our attention was focused on postgeneration optimization—a subject treated only lightly elsewhere. The first criterion was the selection of the condition row with the fewest don't-care values. Beyond that, if a tie had to be broken, the row was selected that had the minimum difference between the number of Y's and N's (again on the advice of Pollack). Later this was compared to the Press method.

It was found that the Press method

- Required more compile time,
- Did not improve object code in the presence of postgeneration optimization, and
- Interfered with "hoisting" optimization steps.

We felt that concentration on postgeneration optimization should be continued, and no further effort was expended on enhancing condition test selection. Clearly, further investigation is warranted. The original selection algorithm is therefore retained. Because an else-column capability was desirable in a decision-table processor, it was provided. The implementation was simply to construct the missing condition rule entries and to supply them with a common action-exit sequence (specified by the user). These added columns are compressed via the merge routine and require no subsequent special treatment. Test selection and hoisting are applied equally to all columns.

Schwayder [12] shows how to incorporate frequency information into test selection. Although this was not incorporated into the processor, the impact on our current generation technique of including Schwayder's algorithm was investigated. If the frequency information specifies the rule frequencies, only one line of APL code need be added to take them into account. Another line of APL code would be required if condition test frequencies are given. More effort would be required to incorporate frequency specification in the decision table format than in making use of it in code generation. Inclusion of optional frequency information is recommended for follow-on work. (As will be described later, some postgeneration optimization methods may destroy some of the effectiveness of test selection.)

The second optimization performed during code generation is that of "hoisting." When one or more actions are to occur in all flow paths following from a single

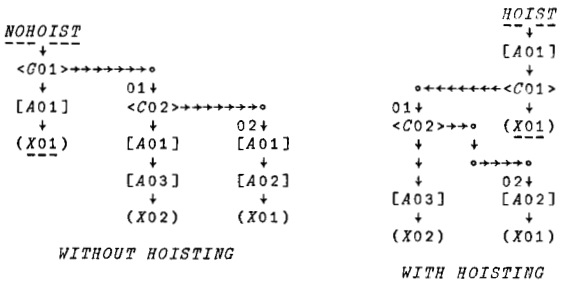


Figure 9 The effects of "hoisting."

condition test, it does not destroy program logic to move (hoist) these actions to a position in front of the condition test. This is a space optimization, because duplicate instances of these actions can be removed from all successor paths to the test. Figure 9 demonstrates the effects of hoisting.

It should be noted that an action cannot be hoisted past another action that must precede it in sequence.

Hoisting can most readily be performed during code generation. Just prior to selection of a condition test, the subtable is examined for an action that:

- Is performed in all remaining rules, and
- Is not required to follow an action that is not performed in all the remaining rules.

Any actions that fulfill these criteria are immediately generated and removed from the subtable.

The action entries accepted by the processor allow the user to indicate an ordering, a *lack of* ordering, or a partial ordering requirement on the actions. This is a degree of freedom not provided in procedural descriptions.

It is not always easy for the user to recognize hoistable actions because he cannot easily recognize the subtables. However, he always has a clear picture of the ordering requirements (or their lack) on the action. Automatic hoisting lets the user specify his logic requirements in terms most easily understood by him.

Unfortunately, hoisting can, on occasion, exert a negative influence on duplicate sequence removal, an optimization procedure that occurs after code generation. This problem is discussed after the description of the affected optimization. But a way has not been discovered to detect the situation without exhaustive (time consuming) combinatorial analysis of the entire table. Fortunately, in the practical examples we have examined, hoisting is more often good than bad. (It seldom had any effect on duplicate sequence removal.) We explicitly resist taking the route of the P.E.T. processor, which requires that the user control this optimization. As mentioned earlier, post-generation hoisting might improve the situation, but was not investigated deeply.

```
003|    C02→16          035|    C02→02
004|    A01             036|    A01
005|    A02             037|    A02
006|02:C03→03          038|16:C03→17
007|    X01             039|    X01
008|03:A07             040|17:A07
009|    C07→25          041|    C07→13

010|    ---→18          042|18:A04
```

Figure 10 Possibly duplicate sequences.

• *Postgeneration optimization*

The two principal postgeneration optimization procedures are *duplicate sequence removal* (DSR) and *duplicate path removal* (DPR). In addition, "scavenger" optimization techniques remove

- Dead (unreferenced) code,
- Redundant condition tests,
- Redundant unconditional branches (branch chains),
- Redundant exits.

The major effect of these optimization procedures is to save space.

• *Duplicate sequence removal (DSR)*

The code generation technique we use assures the performance of all testing required to isolate a rule. If don't-cares are inserted by the user or the merge process, then no tests are performed beyond those actually needed to isolate a rule. Note that rule frequencies are not taken into account, and that a reduced execution time cannot be assured. Average performance can be varied by changing the order of testing, but the minimum amount of testing required to isolate a rule cannot be varied [10]. Beyond reducing the tests to a minimum, one can attempt to restructure the generated code so as to reduce the number of duplicated code sequences. This is done by replacing one of the duplicate sequences with a branch to its equivalent (DSR). In this way a small time loss is introduced (to execute the branch) to save storage space.

DSR is easy to perform because the code generator produces code containing easily noticed patterns of duplicated code. Pairs of code sequences with identical operation codes are first isolated and then more carefully scrutinized (longer sequences first) to assure logical equivalence. If they are logically equivalent, one of them is replaced with a branch to the other. The compared sequences are considered not equivalent for the following reasons:

- They are dissimilar at any point,
- One sequence overlaps a portion of the other, or
- A portion of either sequence has been previously removed (because it was equivalent to some other sequence).

If any of these conditions occur, the unequivalent portions of the sequences are masked off and the remaining "good" portions are individually compared by a recursive procedure. Any duplicated sequence longer than one line of code is removed if it is logically equivalent to another.

Figure 10 shows two possibly duplicate sequences. Note first that lines 010 and 042 are not included in the sequences being compared, but are shown because they are germane to the detailed comparison. Operation codes in lines 003 through 009 match those of lines 035 through 041. The flow paths following these sequences pass to lines 010 and 042, which are seen to be equivalent. The branch paths from lines 003 (035) and 006 (038) are readily compared because they lie within the two sequences being compared. These can be certified as being logically equivalent by comparing the offsets of the branch targets from the beginning of the sequences. Note that in the case of the 003–035 pair the branch targets are across sequences but that this fact does not matter in the comparison. A lengthy analysis must be performed only on the paths emanating from the pair 009–041. This requires a line-by-line comparison of the code starting at label 25 with that starting at label 13.

The cross-link process performed by the P.E.T. processor is an attempt to eliminate duplicate code. Its effect, however, is to eliminate only common trailing portions of action sequences (those that end in an exit). Our algorithms remove all trailing sequences of redundant code, including redundant test trees. In addition, all nontrailing redundant sequences are removed if they flow into logically equivalent code. In general, the results have been very good when applied to actual system programs, as shown by the examples in the Appendix.

It was mentioned earlier that the hoisting optimization can have a negative effect on DSR, because hoisting of an action may remove it from one of a pair of duplicate sequences. When this happens, the pair no longer qualifies for consolidation. It is possible, for example, for hoisting to remove two lines and prevent the removal of, say, ten or twelve lines of duplicated sequences. On the other hand, the removal of an action from a sequence could also cause that sequence to match another, when it would not have done so otherwise. Some cases like this were actually encountered, such as Tables 8 and 9 in the Appendix. Through use of compiler options, selective elimination of hoisting is allowed. Further work should be done to try to establish an effective method for predicting the effect of hoisting on DSR.

Deferment of hoisting until the post-generation phase was considered. Hoisting at that time is much less convenient and will definitely lead to longer compile time. More information must be carried and maintained to keep track of the rule column(s) that were the source of

a particular action. The problem is made more complex by the folding actions of the other optimizers. On the other hand, reasonably good results were obtained with the algorithm used, and we judged the effort-to-payoff ratio too high to implement delayed hoisting.

• *Duplicate path removal (DPR)*

The second major postgeneration optimization technique is duplicate path removal (DPR). When two flow paths that emanate from a condition test have identical leading logic, the leading portion of these paths can be consolidated by moving the condition test down the paths to the point(s) where they differ. Figure 11 illustrates DPR.

As can be seen, DPR may cause reordering of condition testing. The method is to isolate as potential candidates those flow paths emanating from the same condition test and that start with the same operation code. Then a test is selected whose branch target and successor lines contain the same operation code. For each such test, the flow paths are compared and points of difference (POD) are located. (Note that if there are no POD's, the test is redundant and can be removed immediately.) The two paths emanating from the test will be referred to as the "fall-through path" and the "branch path." Conceptually, the process is as follows. Locate the POD's in each of the two paths. Remove the test and place a copy of it just in front of each POD in the fall-through path. The address of each test copied is changed to point to the corresponding POD in the branch path.

With the removal of the original test, it is expected (but not guaranteed) that a large portion of the branch path will become dead code. However, there is no simple way to determine this prior to performing DPR. Therefore a copy of the code is saved before performing DPR. If the code resulting from DPR shows improvement, it replaces the old code. If not the old code is restored. Some time was spent trying to develop some correlation between the amount of improvement and both number of POD's and number of lines in the duplicated paths. However, nothing developed that was useful. Furthermore, the conceptual algorithm described above did not work. This was because DPR was applied after DSR, and DSR could destroy the tree nature of the original unoptimized code. Therefore, DPR has to contend with the interesting possibility that the two flow paths being compared might merge, or cross over. In fact, some cases examined had a single line in the code turn up as POD's in two different paths. This caused the same test to be inserted twice in the same place.

The algorithm finally developed avoids the problems by making a separate copy of the fall-through path, and placing at each POD (in the duplicate) a copy of the original test with the target of the POD in the original

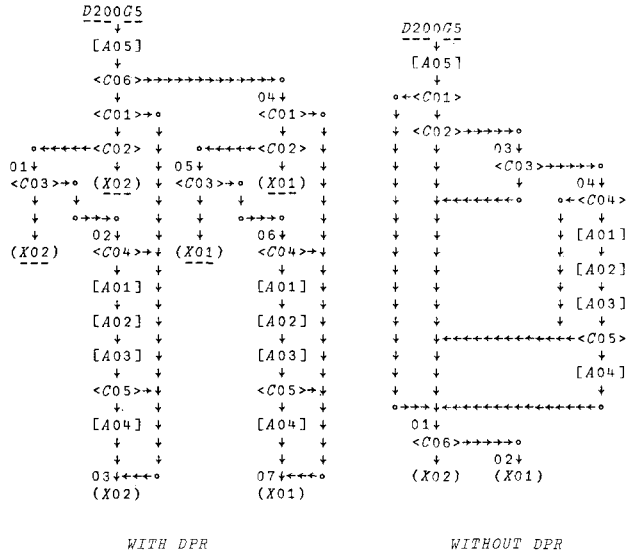


Figure 11 The effects of duplicate path removal.

branch path. This test is followed by an unconditional branch whose target is the POD in the fall-through path. At the end of the path comparison, the original test is replaced with an unconditional branch to the duplicated code, which is placed at the end of the code body. Both the branch and fall-through paths then become candidates for dead code removal.

Because DPR reorders condition testing, it may destroy the effectiveness of optimization procedures that depend upon test ordering. Since DPR never increases storage consumption, only time optimization can be adversely affected. This is a typical time-space trade-off situation. Unfortunately, the user may wish to trade off differently for different points of application of DPR within code from a single table. At present, sufficient information is not carried in the generated code to calculate the overall time costs. A recommended follow-on would be to try to include this.

DPR can recognize only those duplicate paths that emanate from the same condition test. Some tables that were processed contained duplicate paths that did not emanate from the same point and couldn't be removed by the processor. When these paths were removed manually, it was discovered that they could be removed only at the expense of inserting additional tests in the paths. The resulting code then would contain two tests for the same condition in a single path. This is typical of a situation in which the programmer would ordinarily set a switch for later testing. Table 6 in the Appendix contains this type of program structure. No algorithm was discovered that could expeditiously locate duplicate

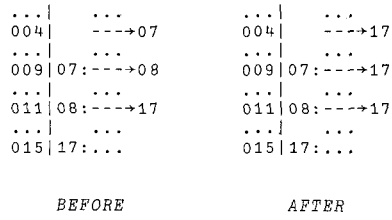


Figure 12 Branch removal.

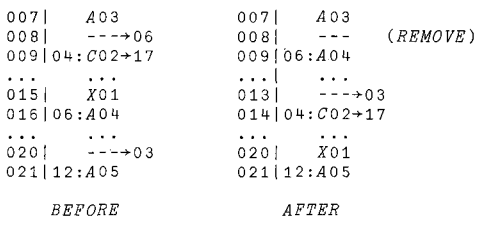


Figure 13 Reordering.

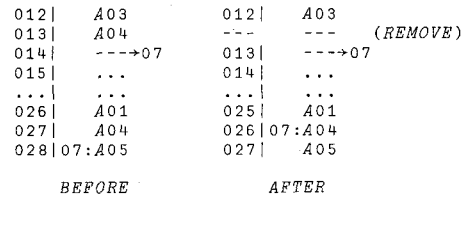


Figure 14 Backward branch movement.

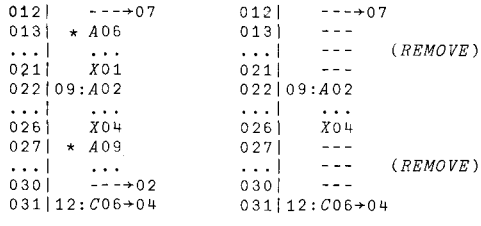


Figure 15 Dead code removal.

paths of this nature. This is also an area that needs further exploration.

• *Scavenger optimization*

Scavenger optimization procedures, as their name implies, clean up the leavings of other optimizers. The scavenger procedures to be described in the paragraphs below are

- Redundant branch removal,
- Redundant exit removal,
- Redundant test removal, and
- Dead code removal.

Redundant branch removal is a procedure for seeking out branches that receive control from other branches and removing them. (Sequences of directly connected branches are called branch chains.) Labeled branches are readily detected. All references (in other instructions) to the label on a branch are replaced with the branch operand, effectively removing the branch from the chain. The labels can then be removed. Figure 12 illustrates branch removal.

The second step is to locate all branches that are preceded by either an unconditional branch or an exit and to remove them. All branches whose targets are exits are replaced with the exits themselves. If a branch has a target that can receive control only from the branch (i.e., the target is preceded by an unconditional branch or an exit), then the branch can be eliminated by reordering the code. This is done, thereby placing the target and all code physically following it (up to a branch or exit) in place of the branch. Figure 13 indicates the method.

While cleaning up branches, this optimizer can perform one additional optimization step, which, strictly speaking, is not redundant branch elimination. It is a type of reverse hoisting. If the predecessor line to a branch target is the same as the predecessor to the branch, the branch predecessor can be removed, and the branch address reduced by one, to refer to the predecessor of the former target. Figure 14 shows this optimization method.

Redundant exit removal is a procedure to remove all exits of any one type that are preceded by a branch or another exit. Care must be taken, however, that there is always one exit of each type left. All conditional branches that have exits as their targets are set so that all references to an exit of one type refer to the same exit. This reduces the requirements for instances of exits to a minimum. (Note that unconditional branches to exits were eliminated by the branch optimizer.) No further processing by the exit optimizer is necessary.

Redundant test removal requires that the flow paths emanating from each conditional branch be examined. If the fall-through path is logically equivalent to the branch path, the test is eliminated. No other clean-up is performed by the test optimizer.

Dead code removal is a procedure that first removes all unreferenced labels. It then locates any unlabeled lines that follow exits or unconditional branches. These

lines are dead and can be removed. Any unlabeled line following a dead line is also dead and can be removed. All dead lines are located and removed at one time. However, because a dead line may have been a branch (conditional or unconditional), the removal of dead lines may give rise to more unreferenced labels. Therefore, the process is iterated until there are no more such labels. Figure 15 illustrates dead code removal.

• *Interaction of the optimizers*
The order of all optimization procedures is:

Pregeneration
    Merge
Generation
    Test selection
    Hoisting
Postgeneration
    Duplicate sequence removal (DSR)
    Scavengers
        Redundant branch removal
        Redundant exit removal
        Redundant test removal
        Dead code removal
        Duplicate path removal (DPR).

The postgeneration optimization procedures are iterative whenever their application can possibly introduce new program structure that would be susceptible to their further application. Further, the scavenger steps are called by both DPR and DSR, and DSR calls DPR. Specifically, DSR processes all of the sequences it can, then calls the scavengers. It then determines whether code was reduced. If so, it repeats. In the case of DPR, after each test instruction is processed, the scavenger procedures are called, followed by DSR. (Calling the scavenger procedures before entering DSR speeds up the latter.) Then DPR checks for code improvement as described earlier.

Concluding remarks

About seventy decision tables taken from actual application areas have been compiled by the system. The code for only two of these could be improved by hand. Furthermore, the compiler running times under the APL system were short enough to allow the experimental model to operate as a production tool. Although there must be manual intervention between processing a decision table and the production of the final code, the processor as it stands has already proved to be a useful aid to some programmers by helping them organize their code. It can be concluded that the processor can be used *now* to gainful ends. If the algorithms described above were recoded into a more fully automated environment, program production could be improved even more.

As has been noted earlier, certain optimization problems are yet unsolved. These are:

- The interference among test-order selection, hoisting, DSR and DPR, and
- Duplicate path removal where the paths do not start at the same condition test.

Some additional optimization procedures could be added to the processor. These are:

- The use of frequency information in test-order selection,
- The use of timing information in conjunction with space/speed priority setting for DPR, and
- The reversal of condition tests.

An investigation of the benefits of post-generation hoisting should also be made.

The optimization methods described in this paper do not include all possible program optimization procedures. We have concentrated on optimization methods that are not usually done by higher-level language processors, namely, the gross arrangement of program flow structure. Specifically ignored are such optimization methods as loop analysis, common subexpression elimination, code motion (except for hoisting and forward code motion), and subsumption [15]. To perform these would require analysis of the information in the decision-table entries, and would require a restriction on the language permitted in the entries. It was felt that a wider service could be performed by providing a framework that would accept any language for entry statements. In this way, optimization procedures that were not performed could still be accomplished by passing the output from the decision-table compiler through another optimizing compiler. Minor revisions to the PL/I printing program can cause it to produce output acceptable to FORTRAN, ALGOL, and COBOL compilers (some of which perform optimization), and even to an assembler macroprocessor.

It should be noted that if a decision table has n rule-columns, then there are exactly n ways of traversing the generated program. This should suggest that exactly n test cases need be prepared to thoroughly test the produced program. Such a test battery would be guaranteed to execute every instruction in the program, and to execute every conditional branch for both yes and no conditions. Because the optimizers fold the program so that some paths through the program execute the (physically) same instructions, it is often possible to completely exercise the program with fewer test cases than the number of rule columns. A test-case generator could easily select a subset of the rule columns that would exercise all

of the code. It is recommended that such a generator be added to the system.

Acknowledgments

The author is indebted to D. H. Manning and P. C. Jacobs for first bringing the problems of decision table compiling to his attention. D. H. Manning, and R. E. Gaiduk are to be thanked for their enlightening discussions on the subject, and our thanks go to May Li and D. H. Manning for providing some "real" decision tables against which the processor was tested. R. H. Williams contributed to the solution of DPR problems.

Appendix: Samples of decision table compilations

This appendix contains decision tables and flow charts of code compiled from them. The reader is invited to browse them to obtain a subjective appraisal of the effectiveness of the compiler. The tables are representative of those examined during the development of the compiler. They were selected to demonstrate several points.

The range of table complexity,
Compile-time range,
The effects of various optimization procedures,
Comparison with other methods,
Some unsolved problem areas.

In addition to these points the reader should also be able to verify that:

There is a unique flow path through the object code for each rule column,
A given flow path contains no redundant tests,
Hoisting situations are not always readily seen in a decision table by the user,
The compiler produced correct code,
The code cannot be improved by hand except where noted.

Table 1 lists the sample decision tables, the time required to compile them on a System/360 Model 50, and the number of lines of object code produced.

In reference to Table 8, note that in some cases the order of condition row selection is arbitrary. When this happens, it is possible that different orderings will produce different amounts of final object code. The condition rules were reversed in D190F3 in Table 8 to create D190F3R. The code produced from D190F3R was 26 lines compared with 20 lines produced from D190F3. Careful examination of the produced code shows that the reversal of selection of conditions 3 and 5 (in the subtable for condition 2 = yes) caused a difference in the hoisting of action 4. When action 4 was hoisted (compiling D190F3), a pair of duplicate paths appeared, one of which was removed by the optimizer.

Table 1 Decision table compile times.

TABLE NAME	TIME (SECS)	LINES OF CODE
CHECK	15	17
CHECKM		15
D190F3	25	20
D190F3R	26	26
D190F4	50	31
D190F4M		26
D190P12	11	9
D200C19	7	7
D200C3	6	4
D200C9	2	3
D200G5	48	15
D200G7	19	18
D200H9	9	8
DTABCD	238	48
FED1	19	23
FED2	19	26
H9PET		10

Table 2 Examples of simple decision tables and their corresponding flow charts.

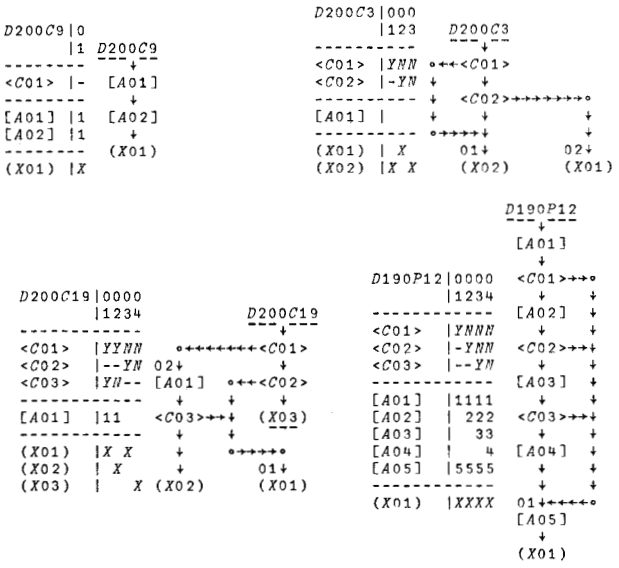


Table 3 A complex table.

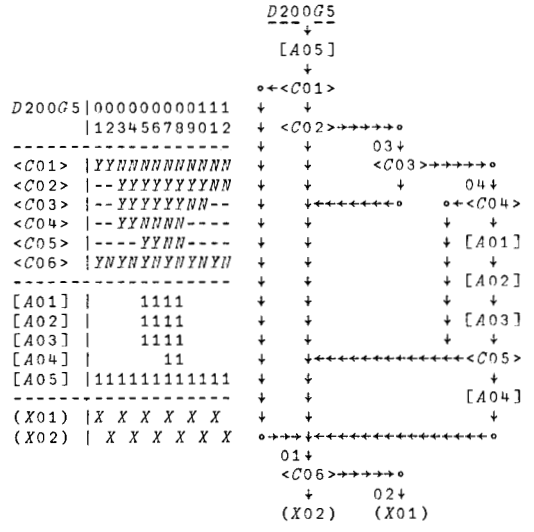


Table 4 Table compiled by our processor and by P.E.T.

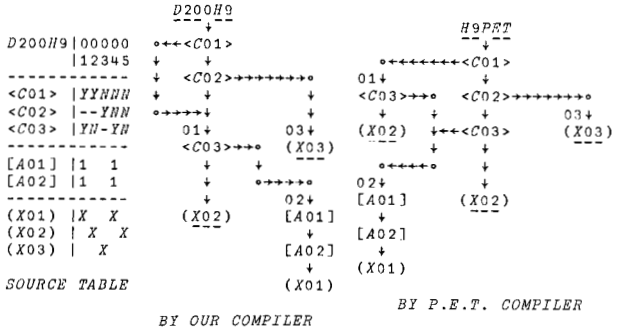


Table 5 A more complex table.

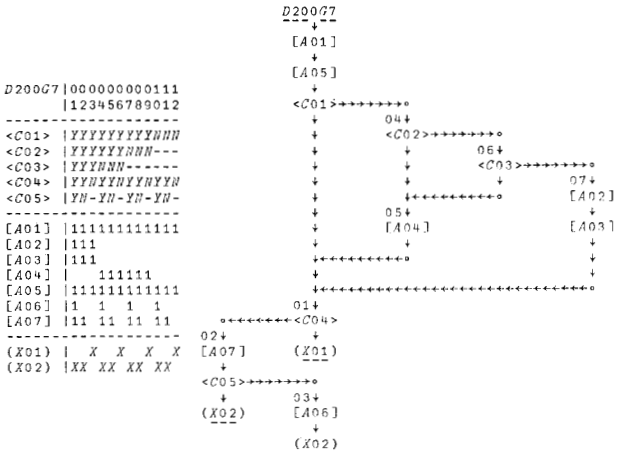
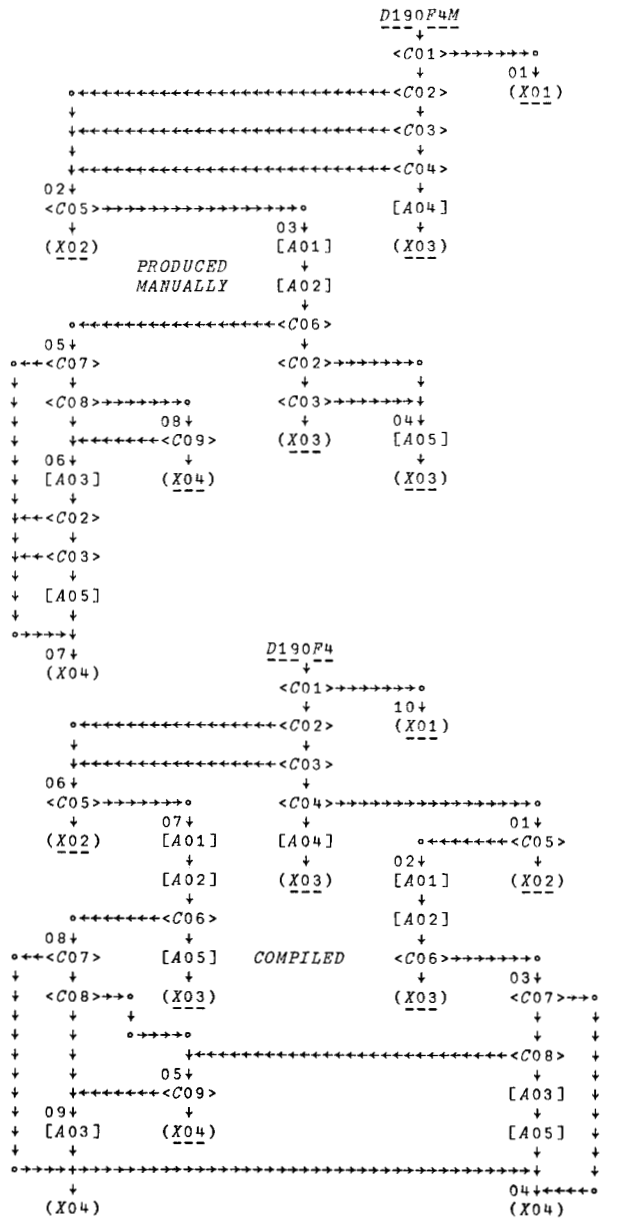


Table 6 Manually improved example.



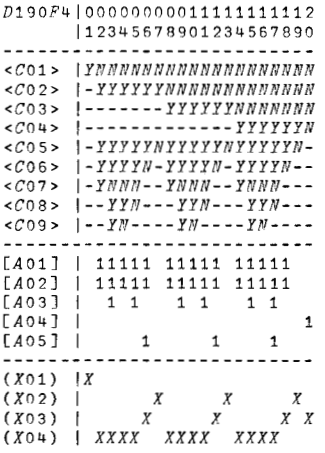


Table 7 Another example of manual improvement.

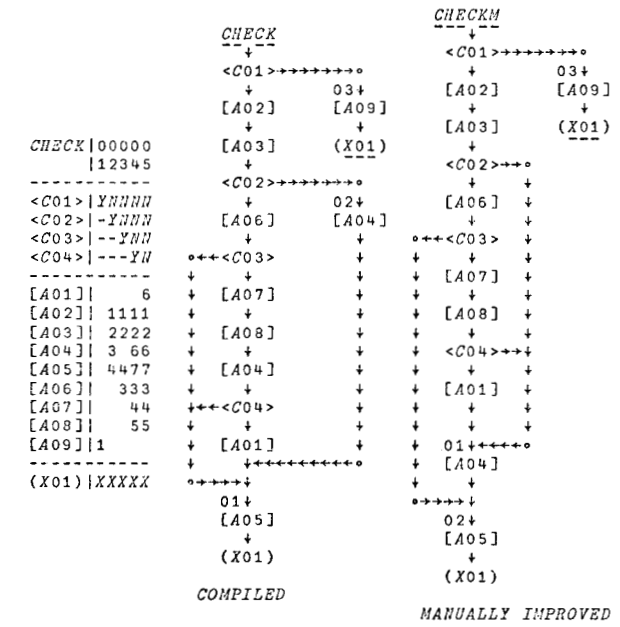
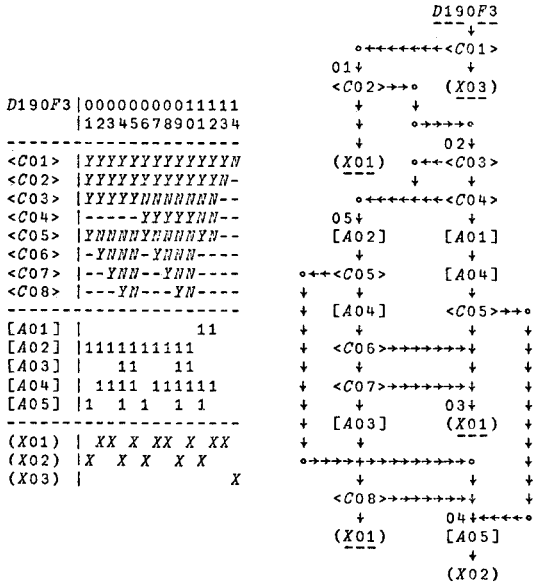
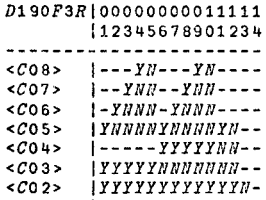


Table 8 Effects of arbitrary condition row selection.





D190F3 WITH CONDITONS REVERSED

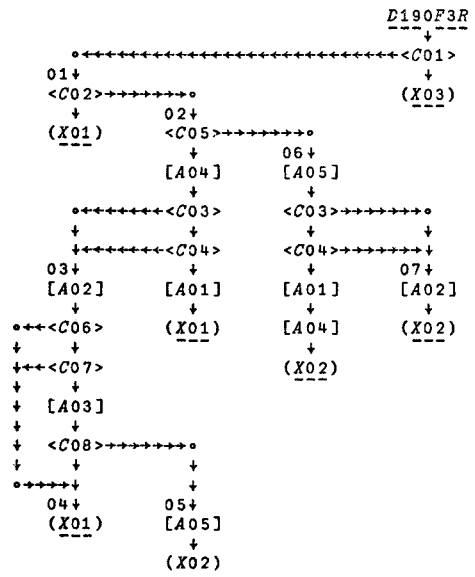
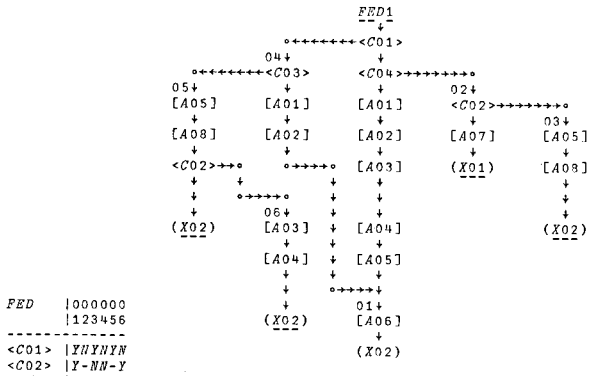
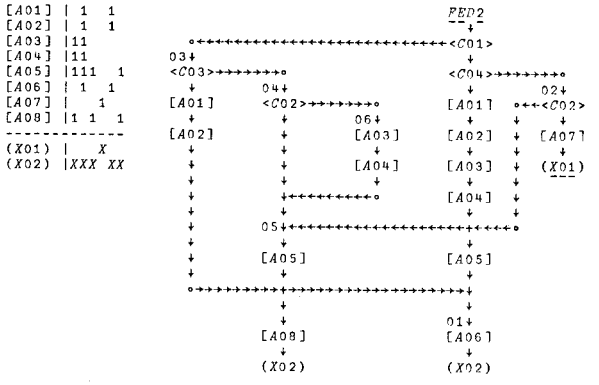


Table 9 Effects of detrimental hoisting.




```
FED     |000000
        |123456
--------------
<C01>  |YNYNYN
<C02>  |Y-NN-Y
<C03>  |Y-Y-N-
<C04>  |-N-Y-Y
--------------
[A01]  | 1  1
[A02]  | 1  1
[A03]  |11
[A04]  |11
[A05]  |111  1
[A06]  | 1  1
[A07]  |   1
[A08]  |1 1  1
--------------
(X01)  |   X
(X02)  |XXX XX
```

23 *LINES COMPILED WITHOUT HOISTING*

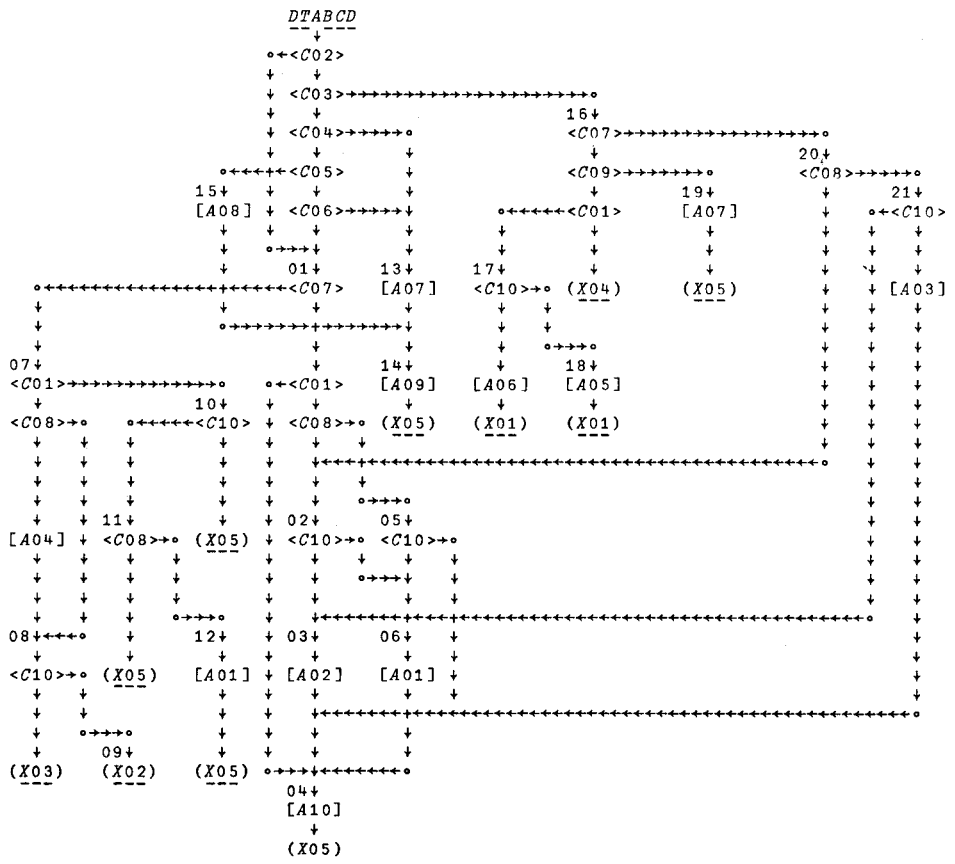


26 *LINES COMPILED WITH HOISTING*

Table 10 A very complex table.

```
DTABCD|00000000011111111112222222222333333
      |12345678901234567890123456789012345
-----------------------------------------
<C01> |YYYY-YY---YYYYNNNNNNNN----NNNNNNNNN
<C02> |YYYYNNNNNNNNNYYYYYYYYNNNNNNNNNNNNN
<C03> |----YYYNNNNNNN--------YYYYYNNNNNNNN
<C04> |-------YNNNNNN------------NNNNNNNN
<C05> |--------YNNNNN------------NNNNNNNN
<C06> |---------YNNNN------------NNNNNNNN
<C07> |YYYNNNN---YYYNYYYYNNNNYYYYNYYYYNNNN
<C08> |YN--------YN--YYNNYYNNYYNN-YYNNYYNN
<C09> |----YNN-------------------N--------
<C10> |YYN--YN---YYN-YNYNYNYNYNYN-YNYNYNYN
-----------------------------------------
[A01] |1          1         11   1        11
[A02] |                       11  1         1
[A03] |                         1
[A04] |                  11             11
[A05] |      1
[A06] |       1
[A07] |     1  1 1
[A08] |        1
[A09] |        111
[A10] |   9          9    99999999     9999
-----------------------------------------
(X01) |     XX
(X02) |               X X           X X
(X03) |                 X X           X X
(X04) |                             X
(X05) |XXXXX  XXXXXXX     XXXXXXXX     XXXX
```

Table 10 A very complex table (continued).



References

1. L. I. Press, "Conversion of Decision Tables to Computer Programs," *Comm. ACM* **8,** No. 6, 385–390 (June 1965).
2. P. J. H. King, "Conversion of Decision Tables to Computer Programs by Rule Mask Techniques," *Comm. ACM* **9,** No. 11, 796–801 (Nov. 1966).
3. H. W. Kirk, "Use of Decision Tables in Computer Programming," *Comm. ACM* **8,** No. 1 41–43 (Jan. 1963).
4. C. R. Muthukishnan and V. Rajaraman, "On the Conversion of Decision Tables to Computer Programs," *Comm. ACM* **6,** No. 13, 347–351 (June 1970).
5. J. F. Egler, "A Procedure for Converting Logic Table Conditions into an Efficient Sequence of Test Instructions," *Comm. ACM* **6,** No. 6, 510–514 (June 1963).
6. M. S. Montalbano, "Tables, Flow Charts, and Program Logic," *IBM Systems Journal* **1,** 51–63 (Sept 1962).
7. S. L. Pollack, "Conversion of Limited Entry Decision Tables to Computer Programs," *Comm. ACM* **8,** No. 11, 677–682 (Nov. 1965).
8. Bell Canada, "P.E.T. (*P*reprocessor for *E*ncoded *T*ables) Processor, Users Manual".
9. H. B. Towne, LTjg USNR, "NAVTABTRANS-C", a computer program, NAVCOSSACT, 1969.
10. L. T. Reinwald and R. M. Soland, "Conversion of Limited Entry Decision Tables to Optimal Computer Programs I: Minimum Average Processing Time," *Journal ACM* **13,** No. 3, 339–358 (July 1966).
11. W. H. Dailey, "Some Notes on Processing Limited Entry Decision Tables," *SIGPLAN Notes* 6, No. 8, 81–89 (Sept 1971).
12. K. Schwayder, "Conversion of Limited-Entry Decision Tables into Computer Programs," *Comm. ACM* **14,** No. 2, 69–73 (Feb. 1971).
13. H. J. Myers and M. Y. Hsiao, "An APL Algorithm for Calculating Boolean Differences," *Proceedings of the IEEE Symposium on Error Recovery Systems* (1969).
14. L. T. Reinwald and R. M. Soland, "Conversion of Limited Entry Decision Tables to Computer Programs II: Minimum Storage Requirement," *Journal ACM* **14,** No. 4, 742–756 (Oct. 1967).
15. E. S. Lowry and C. W. Medlock, "Object Code Optimization," *Comm. ACM* **12,** No. 1, 13–22 (Jan. 1969).

Received February 18, 1972;
Revised April 14, 1972

The author is located at the IBM Systems Development Division Laboratory, San Jose, California 95114.