

Decision  
Tables

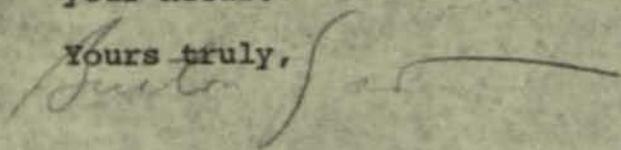
May 28, 1974

Dear Ms. Ritchie:

Enclosed is a copy of "Proceedings of the Decision Tables Symposium" which includes my paper "Structure and Concept of Decision Tables".

Thank you for your interest. I trust this will satisfy your needs.

Yours truly,

  
B. Grad

/rl

Ms. Dagmar Ritchie  
Universitätsbibliothek und TIB  
3 Hannover 1, Welfengarten 1B  
Germany

UNIVERSITÄTSBIBLIOTHEK DER TECHNISCHEN UNIVERSITÄT HANNOVER  
UND TECHNISCHE INFORMATIONSbibliothek

9t  
765-7374

Postanschrift: Universitätsbibliothek und TIB, 3 Hannover 1, Welfengarten 1B

Telefon (Ankunft): (0511) 7622268

Telex: 922168 (tibhn d)

Mr. B. Grad  
IBM Corporation

White Plains, N.Y.  
USA

4/10/74 PM Recalled 2 Boxes

Ihr Zeichen

Ihre Nachricht vom

Unser Zeichen  
(bei Antwort bitte angeben)

1.1.7 Rit

Telefon (Bearbeiter)

(0511) 762-

3 Hannover 1  
Welfengarten 1B

4 April 1974

Dear Mr. Grad,

APR 9 1974

one of our readers is very much interested in your paper  
"Structure and concept of decision tables", held at the Decision Tables  
Symposium, NYC, September 1962.

Unfortunately we have been unable to trace the symposium in any of our  
reference works and any library of this country. We should therefore  
appreciate it very much if you could furnish more detailed information about these  
proceedings of the symposium concerning it such as name and place of  
publisher, as we want to try to obtain the complete volume for our library.

Grad. B., "Decision Tables in Systems Design," *Dig. Tech. Papers, ACM Nat'l. Conf.* 56-7 (Sept. 4-7, 1962), Syracuse, N. Y.  
Grad. B., "Structure and Concept of Decision Tables," *Proc. Decision Tables Symposium, NYC: 19-23 (Sept. 1962)*.  
Grad. B., "Tabular Form in Decision Logic," *Datamation* 7 (7): 22-6 (July 1961).  
Grad. B., "Using Decision Tables for Product Design Engineering," a paper prepared for 1962 AIEE Winter General Meeting, NYC Feb. 2, 1962 (CP 62-378).

Yours faithfully

i.A. *Dagmar Ritchie*  
(Dagmar Ritchie)

All the other above mentioned papers we  
have already in our library.

Decision  
Tables

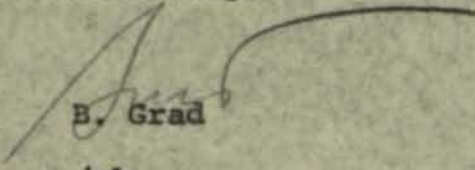
May 28, 1974

Mr. Sol Pollack  
2700 Nielson Way  
Apt. 1336  
Santa Monica, California 90405

Dear Sol,

Thank you for the copy of "Proceedings of the  
Decision Tables Symposium". There was a request  
from the library at Hannover University in Germany.  
I sent them a copy and have kept two for myself.

Sincerely,

  
B. Grad

/rl

VANCOUVER, CANADA

**W** *the Bayshore Inn*

Burt,  
When you've  
made your copy(s),  
please return this  
to SOL POLLACK  
2700 NIELSON WAY  
APT 1336  
SANTA MONICA  
CALIF 90405

IBM

dup  
4/30

International Business Machines Corporation

1133 Westchester Avenue  
White Plains, New York 10604  
914/696-1900

April 15, 1974

3 copies

Mr. Sol Pollack  
1901 Avenue of the Stars  
Suite 880  
Los Angeles, California 90067

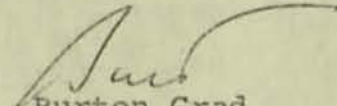
Dear Sol:

It has been a long time since we have talked. I hope everything is going well for you.

Recently I received a request regarding our Decision Table Symposium held in New York City in September of 1962. I have been unable to find a copy of the presentations that were made at that time. I am wondering if you might have an extra one that I could send or, if not, whether you can lend me your copy so that I might make a copy of it.

I'll look forward to hearing from you.

Sincerely,

  
Burton Grad

pm

Attachment

cc: Dagmar Ritchie

DPH 45

Name

Internal Zip

City & State

To: J. M. Wrenn  
Productivity Application Dev.

IBM CONFIDENTIAL

7  
DEC  
TABLE

IBM

DEC 30 1974

From Date: December 20, 1974  
Name & Tie/Ext.: R. N. Purdy X1628  
Title/Dept. Name: Productivity Application Dept.  
Internal Zip/City, State: 1501 California Avenue (69M/037-5)  
or U.S. mail address: Palo Alto, California 94304

Subject: APPLICATION DEVELOPMENT DISCIPLINE (ADD)

Reference: My visit to Los Angeles Scientific Center - December 6, 1974

The purpose of my visit to the L. A. Science Center was to gain some firsthand knowledge regarding ADD, and to consider the current product potentials of Decision Table Techniques. Messrs. Lew Leeburg, ADD Project Manager, and Dave Low presented an ADD overview which included a brief demonstration.

ADD OVERVIEW:

ADD is currently a proposed methodology and architecture for the total replacement of traditional application development cycle functions. Its prime objectives are:

1. Replace current application development mechanisms;
2. Achieve high levels of productivity improvement through
  - revolutionizing end-user to developer communication,
  - enforcing high-quality coding standards,
  - supporting Improved Programming Techniques (IPT) throughout the development cycle;
3. Design from the end-user's and developer's viewpoint rather than that of the vendor or the customer's "DP establishment".

The Science Center project is an Ad-tech prototype aimed at testing the "achievability" of these objectives. ✓

The ADD system would be utilized much earlier in the development process than current mechanisms. ADD would be used in the initial system study phase. It would become the data collection vehicle and communication medium in establishing and maintaining application system requirements.

PADC 11 IBM

The top-down design of an application would evolve as increasing levels of detail are resolved within hierarchies of ADD functional application components. The hierarchies are:

1. Processes - high level structure of functions within a total application system;
2. Blocks - logical segmentations within Processes;
3. Decision Tables - detailed specifications of rules, conditions, and appropriate actions.

Each class of hierarchy represents increasing detail in providing the total application solution.

The additional application components within the ADD functional architecture are:

1. Library procedures - subroutines of procedural or computational code;
2. Message Form - native language dialogs between terminal user and the ADD system or application system;
3. Response Lists - user responses to Messages Forms;
4. Block Linkages - interface specifications between Processes and between Blocks.

The Block Linkages play a key role in ADD's ability to be a data collection and communications medium early in the design process. Block Linkages will provide for "logical binding" between Processes and Blocks. With this binding, the "logical continuity" of the evolving application solution should be able to be tested. The Block Linkage component has not been fully implemented as yet.

ADD embodies a number of variable types. One of the most interesting will be the customizing variable. This variable is initialized to default value(s). The variable can then be customized at numerous stages in the development of an application system. The customizable application system is "executable" throughout the development process--default values are used for unresolved customizing variables.

#### ADD PROSPECTS:

The ADD system is a very interesting proposal for the application development environment of the future. Most of the Application Components will have been prototyped before year-end. The prototype implementation is under VM/370 CMS in APL.SV

The product potential of ADD is long term. The current effort is clearly Ad-tech. The development of the ADD prototype has intentionally ignored all of the aspects

of application data base management and access. The rationale for this is based on the philosophy that application programs should be isolated from application data bases if true data independence, security, and integrity are to be achieved.

Another L.A. Science Center project is addressing the application data base issues. The two prototype efforts are in coordination, and an interface is eventually planned.

If the objectives of ADD are achieved with the prototype effort, I feel ADD may be a useful internal tool for gathering Industry application requirements and initial, solution designs. Industry Marketing MAEP projects would be able to exploit ADD facilities in achieving study objectives.

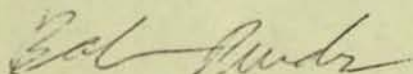
#### PRODUCT POTENTIALS OF DECISION TABLE TECHNIQUES:

The "driving mechanism" of ADD is an exploitation of extended decision table techniques. The logical power of decision tables has been recognized for some time; however, the industry acceptance of these techniques have been disappointing. Some of the reasons are thought to be:

1. Originally, documentation tool only;
2. Tables were awkward to represent and maintain in source card form (fixed length images);
3. Early interpreters were inefficient and generated inefficient code;
4. Entries were limited making the tables large and logically cumbersome;
5. Techniques didn't seem to lend themselves to commercial DP problems; and,
6. Little or no marketing incentive apparent.

The marketing incentives and associated support are really the key issues. Improving application development productivity is clearly growing as an incentive for strategic marketing programs, but without product and market support little real progress can be anticipated. No marketing or market support commitment for Decision Table Techniques exists within DPD. I personally feel the only rationale for aggressively marketing and supporting Decision Table Techniques falls within the scope of Improved Programming Technology (IPT). If it can be agreed that decisions tables are an improved technology, the technique can be addressed by IPT education and marketing programs. ✓

The World Trade PPs, DECTAT for DOS/VS and OS/VS, can be made available within the U.S. with very little effort. Without an aggressive marketing program, it would be unwise to pursue domestic release, however.



R. N. Purdy

cc: L. Seamons, J. Brittain, S. Shaw DPD/HQ, B. Grad DPD/HQ, L.E. Leeburg

700  
12/9

700  
12/18

Call  
Leebing  
w/ Scott Shaw.

Call LA Service Co re  
Decision Tables -

prior to my trip to P.A.

to set up netz in P.A.

reviewed with  
Leebing on 11/26 -  
to present in LA on 12/6 Priority -  
to Scott Shaw + Bob Prady -  
I asked Jim Purvis to send  
either Warren or Jeannette to  
see presentation. ASD 12/2

Call Jim  
Swanson

Spoke to Morris 12/2  
Spoke to Jeannette 12/4

Spoke to Leebing 12/18  
to discuss separate  
AS Proc - CICS?  
PL-1?



To *Mr. Lead*

Memo Slip IBM

Date *10/17/74*

Time *2:40*

Name of person calling

*Mr. Seeburg*

Called to see you  
 Wishes to see you

Please call

Will call again

Returned your call

Tie line Extension Location Area Code Telephone Number Call Back Nbr.

*84545-6316*

For your  Information  Signature  Please  Handle  File  Prepare reply, my signature  
 Comments  Approval  Return  Circulate  See me  And destroy

Remarks

*Call when you get a chance.  
Mr. Henson had asked him to  
Fill you in on the work they  
are doing with Decision Tables.*

From *SL* Phone Location Department Building

To *Mr. Lead*

Memo Slip IBM

Date *10/30/74*

Time *4:10*

Name of person calling

*Lon Seeburg*

Called to see you  
 Wishes to see you

Please call

Will call again

Returned your call

Tie line Extension Location Area Code Telephone Number Call Back Nbr.

*84545-6316*

For your  Information  Signature  Please  Handle  File  Prepare reply, my signature  
 Comments  Approval  Return  Circulate  See me  And destroy

Remarks

*Just Following up. (Henson had  
asked for you two to talk).  
Please call Tomorrow  
OR NEXT Day,*

From *SL* Phone Location Department Building

OCT 15 1974

To: B. Grad 797-TU454 White Plains, N.Y.  
Director of Media and Cross Industry Development

IBM

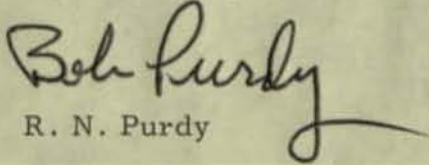
From Date: October 10, 1974  
Name & Tie/Ext.: Robert N. Purdy  
Title/Dept. Name: Productivity Application Development  
Internal Zip/City, State: 1501 California Avenue  
or U.S. mail address: Palo Alto, CA 94304

Subject: Decision Tables

Reference: Your memo to Ms. L. J. Seamons of 10/2/74  
Mr. S.P. Kruzansky memo to H.J. Meyers dated March 25, 1974

Bert, I received the S.P. Kruzansky memo when we were working with the Systems Marketing Productivity Projects Office toward supporting DPD release of DECTAT. Frank Gatewood was very anxious to release DECTAT and/or TABSOL as productivity aids.

Systems Marketing Productivity Marketing and Requirements maintains they have more than enough to do supporting Improved Programming Technologies (IPT) marketing programs without undertaking Decision Tables. I feel Decision tables should be part of IPT; nevertheless, U.S. release of DECTAT is very unlikely for want of aggressive marketing sponsorship. The Industry Marketing response was dismal. We have no other interests in Decision Table Techniques at this time.

  
R. N. Purdy

cc: R. Day, DPHQ, 63W  
L. J. Seamons  
J. M. Wrenn

(F)

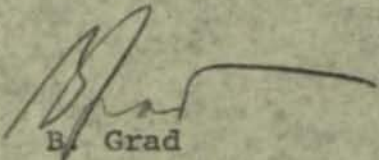
Ms. Lucie J. Seamons  
12 - 63J  
Palo Alto Development Center

October 2, 1974  
B. Grad - xl785  
Director of Media and Cross Industry Development  
Department 797 - TV454  
DPH 1133 Westchester Avenue, White Plains, New York

Decision Tables

Attachment

Is there any interest in Decision Tables in Palo Alto? This was sent to me a few months ago. Frank Dapron was previously in Product Test. I don't think I know Myers.

  
B. Grad

pm

Attachment

}

F. DAPRON suggested that  
you would be interested in  
the first half (at least) of  
this letter.

APR 10 1974 K APR 10

IBM FOK 285-029322

Date: March 25, 1974  
(Div. & location) ISD/MIAD  
(S. mail address): SPD Poughkeepsie  
Dept. & Bldg: 213956  
Line & Tel. Ext.: 255-7573



Subject: EXTENDING DECISION TABLE USAGE TO ALL PHASES

Reference:

To: Mr. H. J. Myers  
DP Science & Marketing Development Center  
D/60A  
2670 Hanover St.  
Palo Alto, California 94304

The most important, yet undersold advantage of Decision Tables is their ability to serve as the basic tool in almost every step of the computer application development process. Specifically:

1. Decision Tables (D. T.) are useful to define and document the external specifications given to the programmer.
2. The programmer/analyst defines the application and assures the completeness of the specifications, via the interactive terminal mathematical verification process of the D.T. compiler. Then, by generating a Flow Chart automatically, he further logically verifies the DT design of the application.
3. The DT Compiler is then used to generate the code from PL/1, Assembler, APL or other languages. This generated code is implicitly well structured modular code.
4. The DT Compiler then identifies an optimal test case library via the TESTGEN functions. (Optimal according to product test criteria of 100% of code execution.)
5. The DT Compiler has the potential to generate the actual input test transaction. Since a DT also defines the actions, some of which are outputs, it should also be able to automatically verify the successful completion of tests.

March 25, 1974  
Extending Decision Table Usage To All Phases  
Mr. H. J. Myers - Palo Alto, California  
Page 2

6. For the maintenance phase, it should be possible to select only relevant regression tests in a minimum test library.

The potential for a DT development technology has not been recognized. TELDAP, which is becoming the FS-SDD modus operandi, has been found to be unnecessary where DTs were used; in fact, the Cause and Effect matrix developed in the course of using TELDAP is a limited form of DT.

Since the 6 previous items describe major processes of the development of a computer program, DTs could be the basis of a Specification Language. A Specification Language, to me, means that if you can define all the specifications for a program in a concise language, you should be able to automate the generation of complex, well-documented, bug-free, implicitly and actually tested computer programs. Though there are small defects in DTs that will need special techniques to compensate for them, the basic economy of reusing the original DT modified to fit the needs of each phase, will lead to faster and cheaper program development.

I hope the designers of Specification Languages will consider the DT format as a basic structure in the language. It appears a better approach than some I've heard.

As we started to discuss a few weeks ago on the telephone, business applications programmers are not interested in testing at the same level as OS developers, Utility developers, or even small scientific programs developers. For business applications, we assume that test variations or rules that concern user input transactions, control cards, or input data, must be better tested than those rules which depend on data already on the data base, (simply because data base information was created by programs not by humans.) Similarly we can assume that system function related rules covering I/O error handling, etc. as shown in the Decision Table stub, should have a lower priority for testing than primary input variations and data base variations. We have, then, for business applications, three categories of tests at least;

- I. Primary user input and easily observable output.
- II. Data base input and output testable functional variations or rules.
- III. System input conditions and intermediate actions which are more difficult to create and/or to observe.

Therefore it should be possible to identify and TESTGEN a weighted set of rules for the selection of tests for various purposes.

1. For Unit Test - 100% path testing all rules, plus a limits testing supplement.
2. For Regression Test - All User input and data base variations generated.
3. For Installation test only valid user interface variations.
4. For a bug correction (APAR, PEAR, CMISTR, ITR) we would like to be able to select all the tests that relate to certain identifiable conditions or actions so that we can avoid a regression on a fix.

To be more specific I would like to see the ability to weight a condition and action in the stub by writing something like:

C8	PRICE = '1212'	;	•40:	Y,N	----	1212
A1	CALC PRICE	;	•10:			1 3
A2	PRINT PRICE	;	•40:			4444

Field indicating weight as a primary input variation.

Then when TESTGENPRIMARY is typed at a terminal - the response could be:

```
11 14 8 7 1 3 5 4 6 9 100% PRIMARY RULES TEST
55% of STUB CNDS & ACTIONS EXERCISED
68% of LEGS OR ACTIONS EXERCISED
```

The response is in the order of greatest weight first. The above assumes a single page DT for the program. There is a more serious problem: How do you identify tests for a whole program not just one decision table? I would like to suggest the following scheme.

Tests should be related for a whole program by program ID and alphabetic ID eg. the most complete definition of a test rule which spans several pages of DTs might be:

E8E7	A 12.	B3.	E4
Load	DT → DT	Rule 3	Rule 4
Module	Page Rule	Page B	on DT
ID	one NO.		page 5
	ID		

March 25, 1974  
 Extending Decision Table Usage To All Phases  
 Mr. H. J. Myers - Palo Alto, California  
Page 4

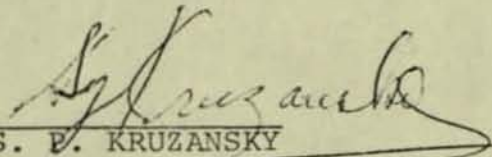
This could mean that if you enter at the terminal

TESTGENPRIMARY	E8E7	the response would be
A11.B4.D3.H2		100% PRIMARY RULES TESTED
A14.B8.F5.G4		55% CNDS & ACTIONS EXERCISED
A8.B9.E2		68% ACTIONS EXERCISED
A7.C6		
A1.C7		
A3.B3		
A5.C		
A4		
A6		
A9		

The sequence of the rules in the list would be the rules with the best test coverage or highest % of actions exercised listed first.

Now that we have combined tests or rules over many DT pages, it would be desirable to select a set of tests that apply to a particular bug or sensitive area in the program.

TESTSELECT E8E7 CND A5 ACT A7 C2 when typed in should respond with a list of rules that affect the identified conditions and actions. The sequence of the list linkage to another page would have an off page call as an Action or as an Exit. An Exit linkage would be a non-returning call or link.

  
 S. P. KRUZANSKY

SPK/gjw

cc: R. Haggerty  
 P. Judge  
 J. Griffin - Sterling Forest  
 F. Dapron - San Jose  
 I. Cutter  
 P. Schlender



To: B. Lead Memo Slip IBM

Date: 1/20

Time: \_\_\_\_\_

Name of person calling: L. Leiber  Called to see you  Please call  
 Wishes to see you  Will call again  
 Returned your call

Tie line: 545 Extension: 6316 Location: \_\_\_\_\_ Area Code: \_\_\_\_\_ Telephone Number: \_\_\_\_\_ Call Back Nbr.: \_\_\_\_\_

For your  Information  Signature  Please  Handle  File  Prepare reply, my signature  
 Comments  Approval  Return  Circulate  See me  And destroy

Remarks: Subj: Possible trip to Palo Alto in Dec.

From: \_\_\_\_\_ Phone: \_\_\_\_\_ Location: \_\_\_\_\_ Department: \_\_\_\_\_ Building: \_\_\_\_\_

To: M. Lead Memo Slip IBM

Date: 11/21/74

Time: 3:00

Name of person calling: \_\_\_\_\_  Called to see you  Please call  
 Wishes to see you  Will call again  
 Returned your call

Tie line: \_\_\_\_\_ Extension: \_\_\_\_\_ Location: \_\_\_\_\_ Area Code: \_\_\_\_\_ Telephone Number: \_\_\_\_\_ Call Back Nbr.: \_\_\_\_\_

For your  Information  Signature  Please  Handle  File  Prepare reply, my signature  
 Comments  Approval  Return  Circulate  See me  And destroy

Remarks: Spoke to Mr. Leiber. Good day for him would be Thurs, Dec 5th. He would probably bring 1 or 2 people with him. Told him we'd get back to him. S. L. - with time you wish if possible.

From: \_\_\_\_\_ Phone: \_\_\_\_\_ Location: \_\_\_\_\_ Department: \_\_\_\_\_ Building: \_\_\_\_\_

To: Lead Memo Slip IBM

Date: 11/28/74

Time: 3:05

Name of person calling: \_\_\_\_\_  Called to see you  Please call  
 Wishes to see you  Will call again  
 Returned your call

Tie line: \_\_\_\_\_ Extension: \_\_\_\_\_ Location: \_\_\_\_\_ Area Code: \_\_\_\_\_ Telephone Number: 84545-6316 Call Back Nbr.: \_\_\_\_\_

For your  Information  Signature  Please  Handle  File  Prepare reply, my signature  
 Comments  Approval  Return  Circulate  See me  And destroy

Remarks: Re: Thurs Palo Alto Mtg.

From: Pat Phone: \_\_\_\_\_ Location: \_\_\_\_\_ Department: \_\_\_\_\_ Building: \_\_\_\_\_

# acm computing surveys:

Volume  
6  
Number  
2  
June 1974

*File  
Dec Tables*

A. V. Aho  
S. C. Johnson

**LR Parsing**

**99**

U. W. Pooch

**Translation of Decision Tables 125**

1082270 00  
R 1 ELLSWORTH JR  
RURAL ROUTE 2  
BOX 310  
BEDFORD  
NY 10506  
SUR  
MAY 28

## REFERENCES

- AHO, A. V., DENNING, P. J., AND ULLMAN, J. D. "Weak and mixed strategy precedence parsing." *J. ACM* 19, 2 (1972), 225-243.
- AHO, A. V., JOHNSON, S. C., AND ULLMAN, J. D. "Deterministic parsing of ambiguous grammars." *Conference Record of ACM Symposium on Principles of Programming Languages* (Oct. 1973), 1-21.
- AHO, A. V., AND PETERSON, T. G. "A minimum distance error-correcting parser for context-free languages." *SIAM J. Computing* 1, 4 (1972) 305-312.
- AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation and Compiling*, Vol. 1, *Parsing*. Prentice-Hall, Englewood Cliffs, N.J., 1972a.
- AHO, A. V., AND ULLMAN, J. D. "Optimization of LR(k) parsers." *J. Computer and System Sciences* 6, 6 (1972b), 573-602.
- AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation, and Compiling*, Vol. 2, *Compiling*. Prentice-Hall, Englewood Cliffs, N.J., 1973a.
- AHO, A. V., AND ULLMAN, J. D. "A technique for speeding up LR(k) parsers." *SIAM J. Computing* 2, 2 (1973b), 106-127.
- ANDERSON, T. *Syntactic analysis of LR(k) languages*. PhD Thesis, Univ. Newcastle-upon-Tyne, Northumberland, England (1972).
- ANDERSON, T., EVE, J., AND HORNING, J. J. "Efficient LR(1) parsers." *Acta Informatica* 2 (1973), 12-39.
- DEMERS, A. "Elimination of single productions and merging nonterminal symbols of LR(1) grammars." Technical Report TR-127, Computer Science Laboratory, Dept. of Electrical Engineering, Princeton Univ., Princeton, N.J., July 1973.
- DEREMER, F. L. "Practical translators for LR(k) languages." Project MAC Report MAC TR-65, MIT, Cambridge, Mass, 1969.
- DEREMER, F. L. "Simple LR(k) grammars." *Comm. ACM* 14, 7 (1971), 453-460.
- EARLEY, J. "An efficient context-free parsing algorithm." *Comm. ACM* 13, 2 (1970), 94-102.
- FELDMAN, J. A., AND GRIES, D. "Translator writing systems." *Comm. ACM* 11, 2 (1968), 77-113.
- FLOYD, R. W. "Syntactic analysis and operator precedence." *J. ACM* 10, 3 (1963), 316-333.
- GRAHAM, S. L., AND RHODES, S. P. "Practical syntactic error recovery in compilers." *Conference Record of ACM Symposium on Principles of Programming Languages* (Oct. 1973), 52-58.
- GRIES, D. *Compiler Construction for Digital Computers*. Wiley, New York, 1971.
- ICHBIAH, J. D., AND MORSE, S. P. "A technique for generating almost optimal Floyd-Evans productions for precedence grammars." *Comm. ACM* 13, 8 (1970), 501-508.
- JAMES, L. R. "A syntax directed error recovery method." Technical Report CSRG-13, Computer Systems Research Group, Univ. Toronto, Toronto, Canada, 1972.
- JOLLIAT, M. L. "On the reduced matrix representation of LR(k) parser tables." PhD Thesis, Univ. Toronto, Toronto, Canada (1973).
- KNUTH, D. E. "On the translation of languages from left to right." *Information and Control* 6 (1965), 607-639.
- KNUTH, D. E. "Top down syntax analysis." *Acta Informatica* 1, 2 (1971), 97-110.
- KORENJAK, A. J. "A practical method of constructing LR(k) processors." *Comm. ACM* 12, 11 (1969), 613-623.
- LALONDE, W. R., LEE, E. S., AND HORNING, J. J. "An LALR(k) parser generator." *Proc. IFIP Congress 71, TA-3*, North-Holland Publishing Co., Amsterdam, the Netherlands (1971), pp. 153-157.
- LEINIUS, P. "Error detection and recovery for syntax directed compiler systems." PhD Thesis, Univ. Wisconsin, Madison, Wisc. (1970).
- LEWIS, P. M., ROSENKRANTZ, D. J., AND STEARNS, R. E. "Attributed translations." *Proc. Fifth Annual ACM Symposium on Theory of Computing* (1973), 160-171.
- LEWIS, P. M., AND STEARNS, R. E. "Syntax directed transduction." *J. ACM* 15, 3 (1968), 464-488.
- MCGRUTHER, T. "An approach to automating syntax error detection, recovery, and correction for LR(k) grammars." Master's Thesis, Naval Postgraduate School, Monterey, Calif., 1972.
- MCKEEMAN, W. M., HORNING, J. J., AND WORMAN, D. B. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, N.J., 1970.
- PAGER, D. "A solution to an open problem by Knuth." *Information and Control* 17 (1970), 462-473.
- PAGER, D. "On the incremental approach to left-to-right parsing." Technical Report PE 238, Information Sciences Program, Univ. Hawaii, Honolulu, Hawaii, 1972a.
- PAGER, D. "A fast left-to-right parser for context-free grammars." Technical Report PE 240, Information Sciences Program, Univ. Hawaii, Honolulu, Hawaii, 1972b.
- PAGER, D. "On eliminating unit productions from LR(k) parsers." Technical Report, Information Sciences Program, Univ. Hawaii, Honolulu, Hawaii, 1974.
- PETERSON, T. G. "Syntax error detection, correction and recovery in parsers." PhD Thesis, Stevens Institute of Technology, Hoboken, N. J., 1972.
- WIRTH, N. "PL360—a programming language for the 360 computers." *J. ACM* 15, 1 (1968), 37-74.
- WIRTH, N., AND WEBER, H. "EULER—a generalization of ALGOL and its formal definition." *Comm. ACM* 9, 1 (1966), 13-23, and 9, 2 (1966), 89-99.

## Translation of Decision Tables

UDO W. POOCH

*Assistant Professor, Industrial Engineering Department, Texas A & M University*

Decomposition and conversion algorithms for translating decision tables are surveyed and contrasted under two broad categories: the mask rule technique and the network technique. Also, decision table structure is briefly covered, including checks for redundancy, contradiction, and completeness; decision table notation and terminology; and decision table types and applications. Extensive literature citations are provided.

*Keywords and Phrases:* decision tables, systems analysis, diagnostic aids, business applications.

*CR categories:* 3.50, 3.59, 4.19, 4.29, 4.49, 8.3.

### 1. INTRODUCTION

The use of decision tables by programmers, analysts, and other users of computer facilities is increasing because they provide a simple tabular representation of complex decision logic. Decision tables, although developed primarily as a vehicle for man-to-man communications, can ease the problems of programming and documentation in many applications where the feasibility of using the traditional flowchart, narrative description, or other communications media is questionable [10, 23, 30, 31, 64, 66, 70, 74, 75, 84, 88, 99, 101].

As higher level programming languages, such as COBOL, FORTRAN, ALGOL, and others, became widely accepted, the communication gap between the computer specialists and the users of computer facilities was expected to disappear. However, this has not been the case, so there continues to be a high degree of misunderstanding in systems analysis and design, and in implementing the chosen procedure into a workable computer program. This is especially true of management-to-man communication, specifically because management frequently does not understand this form of programming language communication. A language form, or structure, is therefore needed to

bridge these man-to-man and man-to-machine communication gaps in these areas. Decision tables can contribute much to improve this communication link (Fergus [29]).

Decision tables provide an effective means of communication between those in and outside the data processing field by defining both the problems and their corresponding logical solutions. In addition, because decision tables succinctly display any conditions that must be satisfied before any prescribed actions will be performed, they are becoming very popular in computer programming and system design as devices for organizing logic, especially when attempting to handle very complex situations, and to be able to account for every possible combination of conditions [23, 32, 57, 62, 66, 89, 92, 104]. Furthermore, the extent and nature of the changes required to update or revise an application program is easily provided by the unique form of the problem statement in decision tables (Auerbach [3]).

Flowcharts, a graphic language form that has also been widely used for man-to-man communications, that was specifically developed for the purpose of representing operations related to computer activities, such

## CONTENTS

- I. Introduction
  - Comparison of Decision Tables and Flowcharts
- II. Decision Table Structure
  - Varieties and Formats of Decision Tables
  - Decision Table Notation
  - Definitions
  - Redundancy, Contradiction, and Completeness
  - Uses and Applications of Decision Tables
- III. Decomposition and Conversion Algorithms
  - Evolution of Decision Table-to-Computer Program Translators
  - Techniques Used in Translating Decision Tables
  - Evolution of Translating Algorithms
  - Scanning and Rule Mask Techniques (Masking Techniques)
    - Rule Mask Algorithm
    - Interrupt Rule Mask Algorithm
  - Conditional Testing and Network Techniques (Tree Structure Techniques)
    - Quick-Rule Algorithm
    - Delayed-Rule Algorithm
  - Ambiguities
  - Automatic Versus Manual Translation
- IV. Conclusion
  - Bibliography

as system analysis, system design, programming, documentation, etc., can also frequently be utilized for other noncomputer-related activities (Chapin [14]).

#### Comparison of Decision Tables and Flowcharts

The decision table is a convenient form for expressing any conditional alternatives, where a particular path to be followed is dictated by a combination of a number of conditions. Flowcharts in such cases can become very complex and difficult to follow, and involve testing for each condition more than once [105].

Decision tables overcome many of the disadvantages of flowcharts as a means of describing computer logic. As may be seen from Tables 1 and 2, decision tables are generally more suitable for direct communication with the computer, and are usually less confusing in the more complex situations, especially if we consider that a decision table contains every possible flowchart which can be drawn for any given problem. Decision tables afford precisely stated logic, more explicit relationships between variables, and simplification of programming (Klick [62]). Thus they provide a convenient way for the analyst or programmer to account for every possible combination of conditions.

It should be noted that the relative merits of decision tables must be weighed against the relative merits of well-structured flowcharts. In other words, with the developing "technology" of structured programming and methods for correctness proof methodologies, the utility of decision tables must be compared with that of a more modern version of programming via flowcharts, rather than with the less disciplined form that was in evidence, especially in nonscientific programming shops, until to very recently. Decisions in a flowchart must be tested in the order in which they appear; however in a decision table (except for the ELSE rule and any specific ordered decompositions, such as the left-to-right decomposition (Harrison [42]) the decision can be tested in any order, depending upon the particular algorithm used in translating the decision table. This enables programmers or

---

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that ACM's copyright notice is given and that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

TABLE 1. ADVANTAGES/DISADVANTAGES OF FLOWCHARTS

<i>Advantages</i>	<i>Disadvantages</i>
<ul style="list-style-type: none"> <li>• Easily produced.</li> <li>• Easily learned (few relatively simple rules and component parts).</li> <li>• Can be used unambiguously to describe the way computers handle data, as well as to represent operations performed by the computer.</li> <li>• Can be produced by computer algorithms from source programs.</li> </ul>	<ul style="list-style-type: none"> <li>• Heavily influenced by personal preference and jargon.</li> <li>• Difficult to follow if the problem conditions are complex.</li> <li>• Revision is difficult.</li> <li>• Limited in displaying all logical elements of the total problem.</li> <li>• Difficult to ascertain if all logical elements are defined and analyzed, especially if the problem conditions are complex.</li> <li>• Flowcharts sharing detailed decision logic are unwieldy, resulting in "macrolizing" difficult sections.</li> </ul>

TABLE 2. ADVANTAGES/DISADVANTAGES OF DECISION TABLES

<i>Advantages</i>	<i>Disadvantages</i>
<ul style="list-style-type: none"> <li>• Clear enumeration of all operations performed.</li> <li>• Clear identification of the sequence of operations.</li> <li>• Easily learned.</li> <li>• Effective means of communication between people in and out of the data processing field; i.e., not limited to computer applications.</li> <li>• Concise and compact form of definition and description suitable for use in analysis, programming, and documentation.</li> <li>• Easy to construct, modify, and read.</li> <li>• Can be used to document applications involving complex interactions of variables.</li> <li>• When applied to computer systems, decision tables foster better use of subroutines, promote efficiency of computer runtime, and provide a complete data check for debugging.</li> <li>• Directly adapted (and possibly converted directly) to computer operations through symbolic logic and computer programs.</li> <li>• Compared with narratives, decision tables are more concise and precise.</li> <li>• Easier visualization of relationships and alternatives.</li> </ul>	<ul style="list-style-type: none"> <li>• For complex situations, they may become extremely large.</li> <li>• Multiple tables may be needed in certain cases to document decision logic (Dixon [23], Fergus [28]).</li> <li>• Many people find the graphic display of flowcharts more meaningful than a tabular description of logic.</li> <li>• Desire for automatic translation ability causes too detailed requirements for man-to-man communication purposes (analogous to the use of programming languages and their restrictions).</li> </ul>

analysts to consider the relative frequency with which transactions satisfy decision rules, and should lead to more efficient programs (Reinwald, et al. [93]). Therefore,

although decision tables are not the answer to all documentation and programming problems, they do offer certain advantages that overcome some of the drawbacks of the

flowchart technique [2, 55, 62, 73, 85, 87, 89, 92, 94, 98]. With the state-of-the-art advancing sufficiently to enable economic conversion of decision tables, their use will show a marked increase.

In Section II a broad spectrum of ideas, including topics on the structure of decision tables, and the varieties and formats of decision tables, are presented. Section III is devoted to the analysis of several different algorithms that can be used for converting decision tables into computer programs. A discussion of the advantages, disadvantages, and ambiguities of these algorithms is given. Finally, it should be pointed out that Shaw [96] and Denolf [20] present extensive, annotated bibliographies on decision tables, and that a recent issue of the SIGPLAN Notices (Shaw [97]) is dedicated completely to various aspects of decision tables.

## II. DECISION TABLE STRUCTURE

A decision table provides a tabular representation of information and data. Information displayed in this manner is easily comprehended by eye, even if the table of information represents a complex logical problem. A decision table is a structure for describing a set of decision rules [4, 9, 13, 28, 46, 47, 49, 57, 68, 72, 103]. The basic structure of a decision table is universally accepted as that illustrated in Figure 1. Although other formats of decision tables exist, some of which are more convenient to certain input/output devices (Pollack, et al. [88]), they are all permutations of this basic format. Decision tables are easy to learn because of their simple structure; and efficiency in using them can be reached with little experience.

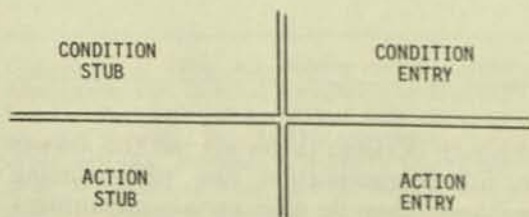


FIG. 1. Decision table structure.

TABLE 3. BASIC ELEMENTS OF A DECISION TABLE

	DECISION RULE 1	DECISION RULE 2	DECISION RULE 3	DECISION RULE 4
IF				
AND				
AND	CONDITION STUB		CONDITION ENTRIES	
AND				
THEN				
AND				
AND	ACTION STUB		ACTION ENTRIES	
AND				
AND				

A decision table can be divided into four quadrants [33, 38, 69, 81, 83, 106, 107]. The upper left quadrant, called the *condition stub*, should contain all those conditions being examined for a particular problem segment. The *condition entry* is the upper right quadrant. These two sections describe the set, or string, of conditions that is to be tested.

The lower left quadrant, called the *action stub*, contains a simple narrative format for all possible actions resulting from the conditions listed above the horizontal line. *Action entries* are given in the lower right quadrant. Appropriate actions resulting from the various combinations of responses to the conditions will be indicated in the action entry. An example of decision rules and the IF-THEN function are illustrated in Table 3.

By considering Table 3, the meaning of the different sections can be illustrated. Each decision rule is a combination of responses to conditions in the condition entry quadrant. The decision rules are numbered for identification purposes in the rule header portion of the table. The topmost horizontal line represents IF, while the remaining horizontal lines represent AND, and the double horizontal line THEN. Note that the condition half of the table is separated from the action half by a double horizontal line and the stub sections are separated from the entry sections by a double vertical line. These lines improve the read-

bility of a table, and can be preprinted on forms. Furthermore, many decision table processors permit ordering on the action entries, thereby making the explicit "AND" of questionable value.

In addition to the decision table elements already discussed, each table usually has a table header. The table header is used for identification purposes when the decision table is processed by the computer. The information that might be found in a table header includes an associated table number, a table name, the type of the table, the number of decision rules, the number of conditions, the number of actions, and various options available to maintain flexibility in formats.

If a condition in the condition stub is true, a Y is entered for that particular rule in the condition entry; if the condition is false, an N would be entered. In a situation where a particular condition is irrelevant, a don't-care would be indicated by use of a dash (-) or an I.

Two other entries, the \* and \$, are used to indicate mutual exclusion of one condition with another on a rule by rule basis (these symbols have been formulated by Pollack, et al. [88] and King [59]). Whenever the case arises within a single rule that the satisfaction of some "required" test (Y or N entry) makes some other required entry a foregone conclusion, then the special entries \* (in place of N) or \$ (in place of Y) can be used to indicate this fact. As an illustration of these inter-condition dependencies, consider the example given in Table 4 (Harrison [42]). Here, 'VALUE' must equal 1 for Rule R1 to be satisfied, at the same time, it may not equal to 3 nor greater than 2. This implies once 'VALUE' = 1 has been determined, the \* will eliminate any further checks on the other two conditions; i.e. they can only be false. In other words, the \* condition is required to be false for that rule, and some other condition for that same rule is adequate to satisfy the requirement. Rule R2, on the other hand, requires that 'VALUE' = 3, and therefore is certainly greater than 2, as indicated by the \$. Thus, the \$ indicates that a condition is required to be true, with some other condi-

TABLE 4. EXAMPLE OF \* AND \$ ENTRIES

STUBS	RULE	ENTRIES
'VALUE' = 1	Y	*
'VALUE' = 3	*	Y
'VALUE' > 2	*	\$

tion available to insure satisfaction of that requirement. A more complete explanation of these entries can be found in Pollack, et al. [88], while an implication of these entries for completeness checking is given in Harrison [42].

#### Varieties and Formats of Decision Tables

Three types of decision tables are in current use today. The *limited entry* table is the most popular and most often used (King [59]). *Extended entry* and *mixed entry* tables are useful in some cases, but because they can always be transformed into limited entry tables, most of this analysis will be concerned with limited entry tables. Examples of the three different types of decision tables are presented in Tables 5A, 5B, and 5C.

In the limited entry table the only allowable entries in the entry quadrants are Y (true), N (false), \* (implicit N), \$ (implicit Y), X (execute action), or I (don't-care), and blank. All of the conditions and actions must be placed in the stub quadrants. Each rule of the decision table should be unique, so logically it does not matter which rule is tested first. Some of the techniques for selecting which rule to test first will be discussed in the next section. Only one rule should be satisfied by a single set of conditions, and if more than one rule can be satisfied the table is said to be *ambiguous* (King [58]).

An extended entry table has part of the condition in the stub quadrant and the remainder of the condition in the entry quadrant. The analogous format applies to the action part of the table. For example, in Table 5B if credit limit is *satisfactory* and pay experience is *favorable*, then approve the order. By considering Table 5B, it can be



## Types of Decision Tables

TABLE 5A. LIMITED ENTRY TABLE

	1	2	3	4
CREDIT LIMIT IS SATISFACTORY	Y	N	N	N
PAY EXPERIENCE IS FAVORABLE	-	Y	N	N
SPECIAL CLEARANCE IS OBTAINED	-	-	Y	N
PERFORM APPROVE ORDER	X	X	X	
GO TO REJECT ORDER				X

TABLE 5B. EXTENDED ENTRY TABLE

	1	2	3
CREDIT LIMIT	SATISFACTORY	UNSATISFACTORY	UNSATISFACTORY
PAY EXPERIENCE	-	FAVORABLE	UNFAVORABLE
SPECIAL CLEARANCE	-	-	NOT OBTAINED
ORDER	APPROVE	APPROVE	REJECT

TABLE 5C. MIXED ENTRY TABLE

	1	2	3
CREDIT LIMIT IS	SATISFACTORY	UNSATISFACTORY	UNSATISFACTORY
PAY EXPERIENCE	-	Y	N
SPECIAL CLEARANCE	-	-	N
PERFORM APPROVE ORDER	X	X	
GO TO REJECT ORDER			X

seen that only one action line is required, whereas in the limited entry table two action lines were required. In general, it can be said that the limited entry table compresses a table vertically, while the extended entry table compresses it horizontally (IBM Corp. [48, 49]).

The mixed entry table is a combination of limited entry rows and extended entry rows (see Table 5C). The PERFORM and GO TO statements in Tables 5A and 5C were not just arbitrarily selected. The PERFORM, as used above, has the same connotation as the PERFORM verb as used in COBOL; i.e., execution is temporarily

transferred into a *closed table* (or a *subroutine*), and control is subsequently returned to the next sequential action of the rule. The GO TO verb is used to exit to an open table or subroutine; that is, no provision is made for control of execution to return to the initiating table (CODASYL [18]). When constructing a table, the GO TO statement should be the last executable action within a decision rule.

## Decision Table Notation

The basic structure presented in the previous section is easy to learn and understand, yet a logical step-by-step analysis is required in the preparation of a complete, accurate decision table. One of the benefits of this tabular method of communication is its adaptability to systematic and analytical techniques for checking completeness, contradictions, and redundancies [8, 11, 25, 26, 43, 44, 52, 53, 88]. Before considering some of the analytical techniques, it is necessary to define some of the notation and terminology in common use.

One of the specific types of Boolean algebra functions is used as the basis for most decision tables. This function, the *AND function*, is considered to be the ordered set of Y, N, I, or blanks that appear in the condition entry boxes for a particular decision rule. The application of the OR function can also be made in decision tables, however this analysis will be limited to the AND function (Hirsehorn [45]). Considering Table 5a, the following AND functions are found:

- the AND function of Rule 1 = YII
- the AND function of Rule 2 = NYI
- the AND function of Rule 3 = NNY
- the AND function of Rule 4 = NNN

To determine whether or not a decision rule is satisfied, evaluate the AND function for that decision rule, and check that it equals the required transaction. For example, the AND function of Rule 3 (NNY) in Table 5a would be the selected decision rule if the transaction was to approve the order, provided special clearance was obtained, even though credit limit and pay experience was unsatisfactory.

### Definitions

Two AND functions are considered to be *dependent* if a transaction exists that satisfies both AND functions. If, on the other hand, a transaction satisfies one, and only one, of the AND functions, that AND function is *independent*.

A *pure AND function* is one that contains no I (don't-cares) (Pollack, et al. [88]). The following is a pure AND function:  $N \cdot N \cdot Y$  (of Rule 3 in Table 56), where "." is defined as the Boolean operator AND.

A decision rule is *simple* if it contains a pure AND function. For example, Rule R3 in Table 5C is simple since it contains the pure AND function  $N \cdot N \cdot Y$ . If an AND function contains one or more I's, it is considered to be a *mixed AND function*. For example the AND function of Rule 2 in Table 5C is  $N \cdot Y \cdot I$ ; hence, Rule 2 is a complex decision rule. If all the decision rules of a decision table are simple, the table is defined as a *full table*; a *partial table* is a decision table that has some mixed decision rules.

### Redundancy, Contradiction, and Completeness

Before discussing the problems of redundancy, contradiction, and completeness, it is necessary to outline two of the basic requirements for decision tables:

- (1) Every decision rule must specify at least one action (weak condition).
- (2) Each transaction must be able to satisfy one, and only one, set of conditions in a decision table. Although there are exceptions to this requirement, for the type of "conventional" tables (Pollack, et al. [88]) under consideration here, this requirement holds (strong condition).

In practice it is often convenient and intuitive to define *all* rules (i.e., no ELSE) implying some no action rules; however, in theory all of these no-action rules should go to the ELSE, therefore the need for Requirement (1). For example, consider the situation where the conditions in a decision rule are: if the customer requests a first-class ticket and a first-class seat is availa-

ble. Without an action, such as "issue a first-class ticket," the above conditions are nonsensical. The second requirement, which is one of the underlying axioms for decision table theory, must be true for other decision table rules to be valid. Compliance with Requirement (2) will also help to insure completeness of decision tables and reduce contradictions and redundancies among decision rules.

Contradictions and redundancies are checked by examining the decision rules to be certain that between each pair of decision rules there exists at least one condition row with a Y, N pair for the two rules. If this Y, N pair does not exist, similar action entries indicate *redundancy*, and different action entries indicate *contradiction* (Pollack [80]). Examples of contradiction and redundancy are illustrated in Table 6.

Rules R1 and R2 of Table 6 are acceptable rules because neither is redundant nor contradictory. However, R2 contradicts R3 and R4 because they all have the same decision rule, yet different action entries. Redundancy exists between R3 and R4 because both have the same decision rule and the same action.

A quick visual check, comparing two decision rules at a time, can easily identify if any redundancy or contradiction exists. If two or more rules do not have at least one Y, N pair in any of the rows, and the actions specified are not identical, then a contradiction of logic exists. An easy way to make this check for redundancy and contradiction is to compare the AND functions of different decision rules. In the following examples the mixed AND functions are

TABLE 6. EXAMPLE OF REDUNDANCY AND CONTRADICTION

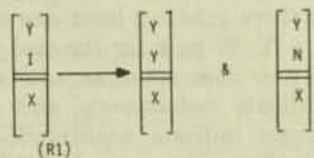
	R1	R2	R3	R4
C1	Y	Y	Y	Y
C2	Y	N	N	N
C3	N	N	N	N
A1	X			
A2		X		
A3			X	X

broken down into pure AND functions to correct the redundancy problem in Table 7.

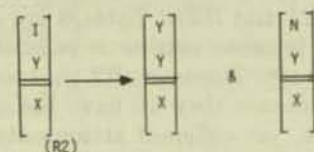
TABLE 7. CREDIT APPROVAL

	R1	R2	R3
CREDIT OK	Y	I	N
PAY EXPERIENCE FAVORABLE	I	Y	N
PERFORM APPROVE ORDER	X	X	
GO TO REJECT ORDER			X

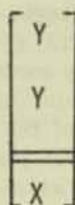
Rule R1 of Table 7 breaks down as follows:



Rule R2 of Table 7 breaks down as follows:



The common AND function of Rules 1 and 2 is:



This AND function can be eliminated. The redundancy-free table is given in Table 8.

TABLE 8. REVISED CREDIT APPROVAL

	R1	R3
CREDIT OK	Y	N
PAY EXPERIENCE FAVORABLE	Y	N
PERFORM APPROVE ORDER	X	
GO TO REJECT ORDER		X

It has been shown how redundancy and contradiction can be checked in decision rules that have both pure and mixed AND

functions. In Table 9 a summary of all the rules for contradiction and redundancy are presented for a pair of decision rules R1 and R2. The AND functions of R1 and R2 are represented by AF1 and AF2, respectively, and the actions are represented by A1 and A2, respectively (Pollack, et al. [88]).

Another problem that always arises is whether or not the decision table is complete, and if it is complete, is there any redundancy, or contradiction in the table. The first step in checking a decision table for completeness is to analyze the table to see if the table contains simple decision rules, complex decision rules (don't-cares), and if any ELSE decision rule is present.

Pollack, Hicks, and Harrison [88] have developed and proved that there exist exactly  $2^n$  independent pure AND functions in a decision table, where  $n$  is the number of conditions found in the decision table. For example, in Table 10A, three ( $n = 3$ ) conditions appear in the decision table, therefore  $2^3 = 8$  possible simple decision rules must exist. All of the decision rules in Table 10A are simple decision rules because there are no "don't care" entries. Furthermore, the decision rules are independent because in any two decision rules, one of the functions contains a Y and the other, an N. Hence, it can now be stated that Table 10a is a complete decision table (Pollack, et al. [88]).

If a transaction in Table 10A is "a guest asking a bartender to mix him a certain type (base) of drink," several rules could be combined. The rules in Table 10B illustrate how Table 10A could be rewritten to contain both simple and complex decision rules. One way to test Table 10B for completeness would be to expand it into Table 10A; however, the following preferred method has been developed (Pollack, et al. [88]):

- Check that each decision rule contains at least one action.
- Check each pair of decision rules to see if they are independent. Do this by checking the AND functions of the decision rules to see that there is a Y in at least one position of the

TABLE 9. CONTRADICTION AND REDUNDANCY

	R1	R2	R3	R4	R5	R6	R7
AF1 AND AF2 ARE DEPENDENT	N	Y	Y	Y	Y	Y	Y
AF1 = AF2	I	Y	N	N	N	N	I
A1 = A2	I	Y	Y	Y	Y	Y	N
AF1 IS PURE	I	I	Y	Y	N	N	I
AF2 IS PURE	I	I	Y	N	Y	N	I
AF1 AND AF2 ARE VALID	X	X		X	X	X	
AF1 AND AF2 ARE NOT VALID							X
REDUNDANT RULE IS AF2		X			X		
REDUNDANT RULE IS AF1				X			
AF1 AND AF2 CONTAIN REDUNDANT RULES		X		X	X	X	
AF1 AND AF2 DO NOT CONTAIN REDUNDANT RULES	X						
AF1 AND AF2 ARE CONTRADICTIONARY							X
AF1 AND AF2 ARE NOT CONTRADICTIONARY	X	X		X	X	X	
THIS RULE IS IMPOSSIBLE			X				

A = ACTION.  
AF = AND FUNCTION.

function and the other function contains a N.

(c) Show that the 4 rules in Table 10B can be expanded into 8 decision rules.

To accomplish (c), recall that each decision rule containing an AND function with an I in *r* positions is equivalent to 2<sup>r</sup> simple decision rules (Pollack, et al. [88]). For example, in Table 10B.

- R1 has 2<sup>1</sup> = 2
- R2 has 2<sup>0</sup> = 1
- R3 has 2<sup>2</sup> = 2
- R4 has 2<sup>3</sup> = 1
- R5 has 2<sup>0</sup> = 1
- R6 has 2<sup>0</sup> = 1

TOTAL 8 SIMPLE DECISION RULES

Table 10B satisfies all of the three tests for completeness, therefore it is a complete decision table.

One way to insure completeness in any decision table is to incorporate the ELSE rule into the decision table. The ELSE rule, by definition, includes all rules not specifically given in the tables. Usually the ELSE rule is merely a convenient catchall rule placed at the extreme right of the table with a special symbol in the rule header that identifies it as such (McDaniel [69]). For decision tables that are converted into computer programs, ELSE rules reduce the amount of needed coding, thereby making the program more efficient. However, the ELSE rule should not be used in this way;

TABLE 10A. EXPANDED BARTENDER

	1	2	3	4	5	6	7	8
BOURBON BASE	Y	Y	Y	N	Y	N	N	N
GIN BASE	Y	Y	N	Y	N	Y	N	N
VODKA BASE	Y	N	Y	Y	N	N	Y	N
OFFER VODKA OR GIN AND VERMOUTH (MARTINI)	X	X	X	X		X	X	
OFFER VODKA AND ORANGE JUICE (SCREWDRIVER)	X		X	X			X	
OFFER VODKA AND TOMATO JUICE (BLOODY MARY)	X		X	X			X	
OFFER BOURBON AND VERMOUTH (MANHATTAN)	X	X	X		X			
OFFER BOURBON AND WATER (OLD FASHION)	X	X	X		X			
OFFER GUEST A SOFT DRINK								X

TABLE 10B. BARTENDER

	1	2	3	4	5	6	7	8
BOURBON BASE	Y	Y	N	Y	N	N	N	
GIN BASE	I	Y	I	N	Y	N		
VODKA BASE	Y	N	Y	N	N	N		
OFFER VODKA OR GIN AND VERMOUTH (MARTINI)	X	X	X		X			
OFFER VODKA AND ORANGE JUICE (SCREWDRIVER)	X		X					
OFFER VODKA AND TOMATO JUICE (BLOODY MARY)	X		X					
OFFER BOURBON AND VERMOUTH (MANHATTAN)	X	X	X		X			
OFFER BOURBON AND WATER (OLD FASHION)	X	X	X		X			
OFFER GUEST A SOFT DRINK								X

it should be used only for those transactions that analysts say cannot possibly happen, and not as a catchall for the convenience of the practitioners. Even with the ELSE rule present, a decision table must still allow ever case to occur. In other words, a table must be checked for exactly how many simple rules are in the table, and that the total number of ELSE rules does not exceed 2<sup>n</sup>. If the total does exceed 2<sup>n</sup>, then a contra-

TABLE 11. THE ELSE RULE

	1	2	3	4	5	ELSE
A = 1	Y	N	1	1	N	
B = 2	1	Y	1	Y	N	
C = 3	1	Y	N	N	N	
D = 4	Y	1	N	Y	Y	
E = 5	N	1	1	Y	Y	
CALCULATE SUM	X	X				
CALCULATE DIFFERENCE			X			
CALCULATE PRODUCT				X	X	

NUMBER OF POSSIBLE SIMPLE RULES

$$= 2^5 = 2^5 = 32$$

NUMBER OF SIMPLE RULES REPRESENTED

$$= 2^r \text{ FOR EACH RULE}$$

$$R1 = 2^2 = 4$$

$$R2 = 2^2 = 4$$

$$R3 = 2^3 = 8$$

$$R4 = 2^1 = 2$$

$$R5 = 2^0 = 1$$

$$19$$

NUMBER OF RULES REPRESENTED BY ELSE RULE

$$= 32 - 19 = 13$$

Conversion of Extended Entry to Limited Entry

diction or redundancy exists. If the total is less than  $2^5$ , it may be readily determined how many simple rules are represented by the ELSE rule (Pollack, et al. [88]). An illustration of the number of decision rules that may be represented by an ELSE rule is provided by Table 11. Since there are five conditions, there are  $2^5 = 32$  possible simple decision rules. The number of simple rules actually present in the table is  $(2^2 + 2^2 + 2^3 + 2^1 + 2^0) = 19$ . The difference, 13 ( $32 - 19$ ), is the number of rules represented by the ELSE rule.

In summary, checking for completeness in decision tables has been discussed with the following three possible constraints:

- 1) Simple decision rules only.
- 2) A decision table with both simple and complex decision rules.
- 3) Decision tables with an ELSE rule.

It should be kept in mind that even though a decision table is a full table, a check for redundancies and contradictions is still a requirement. Once a table has been found to contain no redundancies or contra-

dictions, a check should be made to determine if it is complete.

### Uses and Applications of Decision Tables

Decision tables are useful in many areas of application. This section will analyze some broad-based uses of decision tables, including one specific application.

### In Simulation Models

The previously emphasized advantages of decision tables in handling complex logic makes them a definite aid in formulating the logical flow of simulation models. The queueing structures involved in a model are governed by decision rules which can be easily described by decision table usage. In a model employing decision tables, the table structure is mainly used to determine whether a subprogram is to be executed at a particular time in the simulation. Decision tables also provide a diagnostic aid for the programmer, as well as improving the general communication between the programmer and his problem (Ludwig [66]).

### In an Organization

Decision tables can be used at different levels and for different functions in an organization. Policies of top management may often be expressed tabularly. Tables may be applied to areas such as engineering, mathematics, personnel, and accounting. Furthermore, decision tables can be combined with a decision documentation plan such as that of Fergus [29]. Features of this plan are:

- 1) Use of tables throughout an organization to document all decision making that deserves documentation.
- 2) Tables and their rules are cross-referenced.
- 3) All data elements used in an organization are cataloged and coded.
- 4) All these tables, data element codes, and cross-referencing information are maintained under computer control.

Following this plan permits:

- 1) Display of documented decision making.

- 2) Instant tracing of the effect of decisions throughout the documented structure.
- 3) Rapid reflection and implementation of decision rule changes at all related lower levels.
- 4) Improvement in study and design of large integrated systems while providing a total view of the organization's data flows and requirements.
- 5) Easier application of advanced techniques for systems simulation and information flow studies.

Suggested steps for evolving a decision table usage plan in an organization can be outlined as follows (Fergus [29]):

- 1) Acquire a brief, broad picture of what decision table usage is all about.
- 2) Have at least one individual in the organization become an expert with decision tables.
- 3) Anticipate problems to be incurred through usage.
- 4) Have a reference document.
- 5) Encourage the use of decision tables.
- 6) Explain tables thoroughly to systems users.
- 7) Follow up on the program:
  - a) Identify and correct problems.
  - b) Look for areas that are not using tables, and find out why they are not being used.
  - c) Keep up with developments in the organization that might suggest changes in the use of tables.

#### *In Systematics*

Systematics [40, 41, 56] is a set of techniques for designing and describing information systems which permit the user to concentrate on the design and description of a system without having to consider problems concerned with system implementation.

The basic statement in systematics is called an *element*. The element is actually a special case of a decision table. Three features of this decision table-like element are:

- 1) It is confined to providing rules for obtaining one derivation only. For example, pension contribution and holi-

day entitlement may both be influenced by length of service; the general decision table structure would consider all the relevant states, but the element considers only those which affect one derivation, pensions or holidays, in this case.

- 2) The "primary conditions" determining when an element is performed, are introduced.
- 3) Substitute a "use" list for "go to" information in the action entry of a general decision table form. (This use list contains the names of other elements that use the derivative.)

This variant decision table form is considered to be more manageable because the entries are limited to the combinations of conditions that yield the derivation of only one item. The new table is output-oriented in that the designer can work back through the output and determine exactly which rules have been used for a particular derivation. The new table form also avoids any processing sequence, and therefore permits directing attention to any one element, ignoring the rest of the system (Grindley [40]).

#### *In Automatic Test Equipment Systems*

The use of a programming language, based on decision table techniques, permits the test engineer to write test statements easily, and permits programming a test specification with minimal knowledge of programming techniques and of the specific test equipment system involved.

The envisaged program involves the process of translating test requirements into a test program. A testing system must automatically perform any sequence of tests on a unit being tested, and must choose a new sequence of tests in accordance with previous results.

A modified decision table structure is used with the test conditions placed in the condition statements quadrant of the table, with the resultant test actions being placed in the action statements quadrant and the necessary testing parameters filling in the rules portion. The advantages gained by the

decision table structure are, once again, to enable the test engineer to divorce himself from both knowledge of programming techniques and the test equipment itself.

### III. DECOMPOSITION AND CONVERSION ALGORITHMS

#### Evolution of Decision Table-to-Computer Program Translators

Systems analysts and individuals involved in program development were confronted with situations in which existing methods of problem descriptions, such as flowcharting and narratives, were inadequate. As a result, decision tables were invented (McDaniel [67], Pollack [84]). Truth tables like that in Figure 2 and logic tables, such as Federal Income Tax forms, had existed prior to the introduction of decision tables; but they did not have a standard format, nor were they automatically convertible into computer programs (Quine [90, 91]). The logic used in those truth tables provided a ready springboard for pioneers in the development of decision tables. The truth table in Figure 2 indicates the truth values that X and Y can assume, as well as the truth values of the logical statements "X OR Y" and "X AND Y". An up-to-date discussion of the application of decision tables to tax forms is given by Ainslie and Kenney [1].

In 1957 a task group at General Electric, one of the earliest users of decision-structured tables, developed such tables, and a computerized method of solving them. The processor for solving these tables initially operated on an IBM-702, and was subsequently implemented on the IBM-305, 650, and 704. An improved processor and language called TABSOL (Tabular Systems

X	Y	$A \vee B$	$A \wedge B$
T	T	T	T
T	F	T	F
F	T	T	F
F	F	F	F

FIG. 2. Truth table.

ITEM-A	ITEM-B	ITEM-C	THEN GO TO
EQ 3	NE 4	EQ 20	TABLE-2
NE 3	EQ 10	EQ 40	TABLE-3
EQ 5	NE 4	EQ 60	TABLE-4
NE 5	EQ 10	EQ 80	TABLE-5

FIG. 3. Horizontal rule format (TABSOL).

Oriented Language) was implemented on the GE-225 in early 1961 [35, 50, 51, 63, 88]. With the TABSOL processor, an analyst could bypass the programmer, and input the prepared decision table directly to the compiler for processing. This processor, which is still in use, utilizes the horizontal rule format, in which the decision rules are read horizontally. For example, the first decision rule of Figure 3 reads: If Item-A EQ 3 and Item-B NE 4 and Item C EQ 20 then GO TO TABLE-2.

About the same time, a special committee for the CODASYL (COntference on DATA SYstems Languages) Group, studying decision tables, developed a decision table language known as DETAB-X (Decision Tables, Experimental) [7, 16, 17, 19, 69, 77, 78, 79, 80, 82, 86]. In 1965, a follow-up by a working group of the Los Angeles ACM SIGPLAN (Special Interest Group for Programming Languages) resulted in an improved processor called DETAB-65. This processor, integrated into a COBOL compiler, was a significant data processing development [82, 95, 104].

Other decision table innovators in the early 1960s were Hunt Foods and the Sutherland Company. About this time, work by IBM in conjunction with the Rand Corporation, resulted in a processor called FORTAB, using the scientific programming language FORTRAN (Fergus [27]). Prior to FORTAB, most of the processors had used COBOL. The Insurance Company of North America, produced a decision table processor called LOBLOC which was used on the IBM-7080 [21, 22, 36, 37].

Several government agencies also participated in this early development. CENTAB was developed through a joint effort of the United States Bureau of the Census and Sperry Rand. A processor subsequent to CENTAB was TAB-7C which, like FORTAB,

used FORTRAN as its base programming language.

One of the more unique processors developed during the 1960s called PET (Preprocessor of Encoded Tables) was a product of Bell of Canada. PET, using a PL/I decision table language program, produced PL/I source statements (Fergus [27]). Some of the more recently developed processors include DETRAN, DETOC, DETAP, DETAB-70 (McDaniel [68]), TABTRAN, SMP, DECISUS (Pollack, et al. [88]), and LOGTAB (King [54]), with an application in systematics [40, 41, 56].

Many favorable comments have come from the users of decision tables. The supervisor of preparation of computer programs for the 1964 Census of Agriculture stated that the very extensive and comprehensive consistency checks and the resulting desirable adjustments in data could not have been computerized without the use of decision tables (McDaniel [69]).

An extremely complex file maintenance problem arose at the USAF ARLS (Automatic Resupply Logistic System) at Norton AFB, Calif. Almost seven man/years had been spent trying to define the problem using narrative descriptions and flowcharts, but to little avail. Then a crash program using decision tables was implemented. Four analysts spent one week establishing the decision table format. Three weeks later the problem was completed (Fisher [32]).

#### Techniques Used in Translating Decision Tables

The purpose of this section is to discuss the translation of decision tables into computer programs. For this purpose, an orderly procedure, or algorithm, is needed to translate the tabular structure of the decision table into an efficient sequence of machine-executable instructions. The term *decomposition* is used to describe any of the techniques by which decision tables are converted into conventional decision trees or programming language (Pollack, et al. [88]). Two main classes of decision table-to-computer program conversion algorithms are available: tree, or network, structure methods [71, 85, 92, 94, 98] and mask methods [55, 62, 89]. The algorithm selection

should be based on efficiency criteria established by the user [2, 12, 24, 73, 76, 85, 87, 89, 92, 94, 98, 108].

Efficiency of a decomposition algorithm usually falls into one of two classifications; namely minimization of storage necessary for the object program (Pollack [85], Sprague [100]) and minimization of processing time (King [55], Montalbano [71]). It is impossible, with current techniques, to design an algorithm that would be globally optimal in all situations, so it is necessary to analyze the constraints in each situation before determining what kind of an algorithm to consider. Once an algorithm has been selected, it can either be used in a hand coding technique or built into a preprocessor compiler for automatic translation of tables (Pollack, et al. [88]).

Most of the decomposition algorithms deal with limited entry decision tables rather than the extended entry or the mixed entry decision tables [85, 92, 94, 98]. This does not create a problem because extended and mixed entry decision tables can be easily changed into limited entry decision tables. An extended entry table is illustrated in Table 12a and an equivalent limited entry version in Table 12b.

There are two major ways of translating a limited entry table into a computer program. The first technique, called *scanning* (or, at more sophisticated levels, the *rule mask technique* [5, 28, 55, 105]), involves

TABLE 12A. ORIGINAL EXTENDED ENTRY

	1	2	3	4	ELSE
X =	6	2	4	4	
Y =	1	3	1	3	
COMPUTE Z =	X - Y	X + Y	X - Y	X + Y	

TABLE 12B. NEW LIMITED ENTRY

	1	2	3	4	ELSE
X = 2	*	Y	*	*	
X = 4	*	*	Y	Y	
X = 6	Y	*	*	*	
Y = 1	Y	*	Y	*	
Y = 3	*	Y	*	Y	
COMPUTE Z = X + Y		X		X	
COMPUTE Z = X - Y	X		X		



testing each transaction against all pertinent conditions in a single rule and scanning across the rules until one is found in which all conditions are satisfied. The solution is said to be in that rule, and consequently the actions associated with this rule are executed. To minimize run-time for the resultant program the rules are often ordered on the frequency in which they are expected to be selected (Fergus [28], Pollack, et al. [88]). An example of the scanning technique is illustrated in Figure 4, using Table 13 as a sample decision table.

#### Scanning Technique

TABLE 13. SAMPLE SCANNING TABLE

	1	2	3	4	ELSE
C1	Y	N	Y	N	
C2	N	-	Y	N	
C3	Y	Y	-	N	
A1	X	X	X	X	

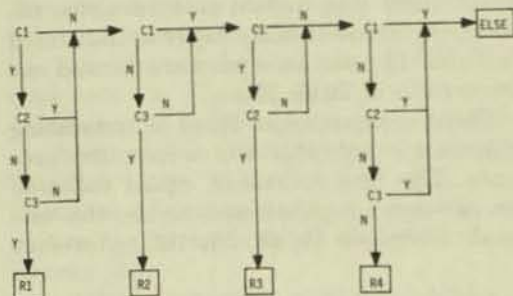


FIG. 4. Testing sequence of sample table.

The transaction vector is compared with one condition at a time in each rule. For example, test the table at the first pertinent condition and if the first condition is satisfied, test the transaction entry against the first pertinent condition of the second rule. The complete scanning technique of Table 13 appears in Figure 4. (Note that Table 12 contains mutually exclusive conditions such that, as soon as "Y = 1" has been determined, no further checks on the "Y" values need be made. The \* notation eliminates,

for example, the "AND" function  $C1 \cdot C2 \cdot C3 \cdot C4 \cdot C5$ .)

The second technique for translating limited entry decision tables into computer programs is called *condition testing*, or the *network technique* (Fergus [28], Montalbano [71]). This method tests one condition at a time and requires the rules in the decision table to be unambiguous; i.e., one transaction cannot satisfy more than one rule. The network technique takes advantage of this requirement and seeks to isolate the unique rule satisfied by each transaction entry. It is primarily condition-oriented (Pollack, et al. [88]). A typical decomposition tree for a decision table is given in Figure 5 (Fife [31]).

It can be observed from Figure 5 that one row of the original decision has been selected as the starting point. The particular row that is selected for a starting point can be based on several different criteria that are discussed later. The condition in the selected row becomes the first comparison in the tree structure. The original decision table is then decomposed into two subtables

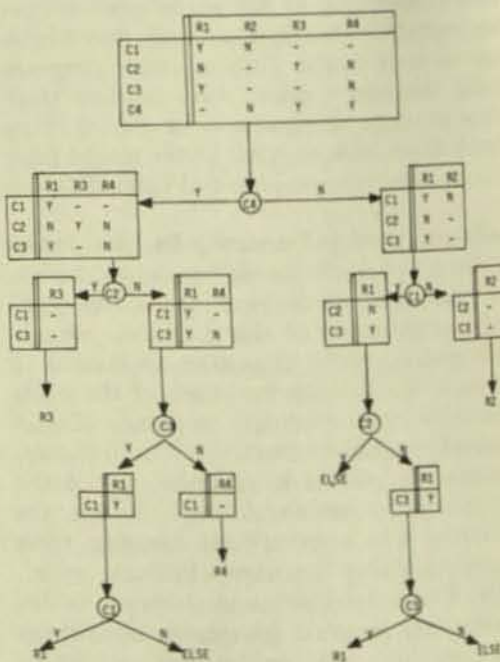


FIG. 5. Decomposition tree of a decision tree.

(containing one less row), one subtable and one rule, or two rules; each of these is associated with each branch of the comparison. A row is then selected from each of these subtables and its condition is tested. This process is continued until each rule of the original decision table or an ELSE rule appears as one of the branches of a condition (Pollack [85]).

In dealing with these specifics of the two previously mentioned decomposition techniques, the action part of the decision table is omitted because the algorithms are designed to isolate a unique rule which, in turn, defines an action set. (Only limited entry tables are discussed because the extended entry decision tables can be converted to limited entry tables.)

#### Evolution of Translating Algorithms

Most of the decision table techniques discussed in the literature, as was seen in the previous section, can be divided into two broad categories. Techniques that optimize core storage, and those that optimize execution time; some techniques attempt to optimize both categories.

Montalbano [71], the first to devise techniques for obtaining computer programs to optimize storage requirements and execution time, developed two methods called the *Quick Rule Method* and the *Delayed Rule Method*. The objective of the quick-rule method is to perform, as soon as possible, those tests which will isolate a rule as quickly as possible. This method is efficient with respect to storage requirements. The objective of the delayed-rule method is to delay the tests isolating rules as long as possible. This method is efficient with respect to average execution time. Montalbano's work was used as a basis for the techniques developed by Pollack, [85].

The objective of Pollack's first algorithm is to convert a decision table to a computer program using the minimum number of storage locations. In his second algorithm, the objective is to convert a decision table to a computer program in which comparisons can be executed in minimum time (Pollack [85]). The algorithms automati-

cally handle the ELSE rule and isolate any redundant or contradictory decision rules during the conversion process.

A different process that used a rule mask technique was developed by Kirk [62]. This technique resulted in the optimization of storage requirements, but was inefficient in average execution time because it required the sensing of all conditions by way of a mask which is used to screen out nonpertinent conditions according to the input data prior to scanning the decision rules. Further work in this area was done by Press [89] whose method offered better run time optimization than Kirk's technique. Another technique that expanded Kirk's work was developed by King [55]. One of the assumptions in this technique is that advance information on evaluation times and frequency of occurrence of rules is available. King's method offers a marked savings in computer run time in comparison with Kirk's, but it uses more core storage space because of the increased complexity of the branching structure.

Some techniques for programming decision tables in higher level languages were explored by Bjork [6] and Veinott [104]. Specifically, they used FORTRAN, COBOL, and ALGOL in their translation of decision tables to programs. One of the most rigorous works on translating decision tables into an optimal branching sequence has been done by Reinwald and Soland [92, 94]. They have developed two algorithms that minimize run time and core storage plus optimizing the resulting test sequence. Furthermore, they claim that the two algorithms can be combined to yield a testing sequence that minimizes the total cost of both core usage and run time (Pollack, et al. [88]). Even though these algorithms are quite efficient, they are not widely used because of their complexity. Besides they require prior information concerning the frequency with which the decision rules are satisfied.

Further work on Pollack's algorithm has been done by Shwayder [98]. He proposed two alternatives to Pollack's algorithm which he advises will result in lower execution time. His first alternative uses the

communications concept of entropy (i.e., a measure of the variability of a set of messages) and Shannon's noiseless coding theorem. This algorithm is most effective when the estimated frequency of the ELSE rule is very low. Shannon's noiseless coding theorem can be used to find the average code-word length, which is necessary in order to minimize average code-word length. Shwayder's second modification completely tests the ELSE rule, but results in greater run time. These alternatives do not necessarily lead to globally optimal solutions because they suboptimize one subdecision table at a time.

A technique for parsing large decision tables into smaller ones is offered by Chapin [12, 14] who developed a technique whereby a decision table can shrink to one twenty-fourth of its original size by use of parsing methods. Another parsing technique, proposed by H. Strunz [102], permits parsing utilizing only the syntactical characteristic of the decision problem. It requires a description of the problem in decision grid chart format, and allows the development of decision tables within defined limits by avoiding, or at least minimizing, repetition of conditions and actions in the resulting tables. Some of the factors affecting decision table parsing are indicated in Figure 6.

Hierarchies for different levels of decision tables can be established by using the interrelationships of Figure 6. An example of vertical parsing would be separate tables dealing with data at various levels such as

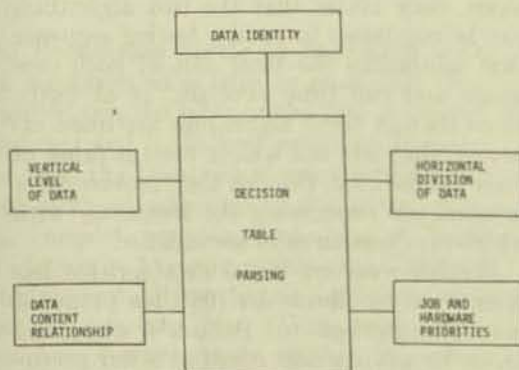


FIG. 6. Parsing of decision tables.

file, volume, record, field, character, and bit level. Parsing of data to recognize horizontal structure would utilize separate tables for head, body, and tail of the data sets. Job and hardware priorities would depend on the type of environment in which the decision table is processed.

Data content can be tested or sorted, and then grouped into separate decision tables according to content. The parsing factors shown in Figure 6 can be given different priorities, depending upon the type of processing environment.

Another method of parsing the tables is the use of proper, or more effective, linkage between tables. It is possible, in an action entry in one decision table to direct entry into another table. If the new table is entered without qualifications, then it must be processed from the beginning. If the directive statement is actually a return to a particular rule in a table from which an exit was originally made, then it can be said that the task has been broken into parts; that is, instead of the complete processing of each table, only parts of each table may be necessary to process and satisfy input data (Chapin [12]).

#### Scanning and Rule Mask Techniques (Masking Techniques)

The straight scanning technique, which has already been discussed, is inefficient with respect to the utilization of core storage and run time. This technique has no remembering capability in its testing sequence, so the same condition may be interrogated many times. One way to improve scanning is by using the rule mask technique (Barnard [5], Kirk [62]).

#### Rule Mask Algorithm

Many of the authors refer directly to Kirk's article [62] and his algorithm, therefore a fairly detailed outline of this algorithm is given:

- 1) Prepare a binary image of the condition matrix of the table by placing a "1" in each position in which the original table has a "Y" and a "0" in all

other positions. Table 14 is the original Credit Approval decision table, and Table 15 shows the *table matrix* for Table 14.

TABLE 14. CREDIT APPROVAL

	R1	R2	R3	R4
CREDIT LIMIT OK	Y	N	N	N
PAY EXPERIENCE FAVORABLE	-	Y	N	N
SPECIAL CLEARANCE OBTAINED	-	-	Y	N
DO APPROVE ORDER	X	X	X	
DON'T APPROVE ORDER				X

TABLE 15. TABLE MATRIX FOR CREDIT APPROVAL

	R1	R2	R3	R4
C1	1	0	0	0
C2	0	1	0	0
C3	0	0	1	0

- 2) A masking matrix is needed to screen out nonpertinent conditions from the transaction or data vector prior to scanning the table matrix. A *masking matrix* is made by placing a "1" wherever the original decision table shows a pertinent condition ("Y" or "N"); everything else is set to zero. Table 16 shows the masking matrix for Table 14.

TABLE 16. MASKING MATRIX FOR CREDIT APPROVAL

	R1	R2	R3	R4
C1	1	1	1	1
C2	0	1	1	1
C3	0	0	1	1

- 3) Prepare a binary transaction vector by placing a "1" in each true condition position and a "0" in all other positions. A simple transaction entry and its vector is shown in Figure 7.

CONDITION 1 - FALSE	$\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$
CONDITION 2 - TRUE	
CONDITION 3 - TRUE	

Fig. 7. Transaction vector.

- 4) The actual scanning operation is made rule by rule. The first rule of the masking matrix is logically multiplied by the transaction vector to eliminate the rule's nonpertinent conditions from the transaction vector. The result is then compared with the first rule vector of the table matrix. If the two are equal, the rule is satisfied. If not, the scan proceeds to the next rule. Table 17 illustrates the scanning operation, and it indicates that Rule 2 satisfies the transaction entry.

TABLE 17. SCANNING OPERATION

RULE	DATA VECTOR	"AND"	MASKING VECTOR	RESULT	TABLE VECTOR	EQUAL?
1	0	X	1	+ 0	1	NO
	1	X	0	= 0	0	
	1	X	0	= 0	0	
2	0	X	1	= 0	0	YES
	1	X	1	= 1	1	
	1	X	0	= 0	0	

In terms of total storage requirements this approach appears to be very efficient. Each test need appear only once in the program; and additional storage required to generate and interpret the mask may not be much greater than that used for transfer instructions to achieve the branching inherent in the conditional testing techniques.

With respect to average processing time, however, this approach is not very efficient, since all conditions must be tested regardless of the nature of the input, and additional time must be spent generating and interpreting the mask (Reinwald [92, 94]). A method for adding some improvements to the rule mask technique is the interrupt rule mask method.

#### Interrupt Rule Mask Algorithm

One of the drawbacks to the rule mask technique, as presented above, is that it might produce object programs of longer run time than necessary (King [55]). A modification of the rule mask technique, discussed below, takes into account both

rule frequencies and relative times for evaluating conditions. The interrupted rule mask procedure, due to King [55], does not evaluate the rules of a decision table in a sequential manner like the rule mask technique. Before discussing the strategies for interrupting the testing sequence, a few terms need to be defined; therefore, let

$T$  = expected execution time for a program;

$t_i$  = evaluation time for each condition;

$f_j$  = frequency of occurrence for each rule;

$S$  = time for carrying out the testing of transaction vector for a single rule;

$\Sigma f_j$  = total frequency.

Some of the above conditions must be determined or estimated for the decision table under consideration. The total run time (see Table 18) can be determined for the simple rule mask technique by using the following formula:

$$T = (t_1 + t_2 + t_3 + S)f_1 + (t_1 + t_2 + t_3 + 2S)f_4 + (t_1 + t_2 + t_3 + 3S)f_3 + (t_1 + t_2 + t_3 + 4S)f_2$$

Testing according to frequency of occurrence, and substituting the values of Table 18 into the formula gives:

$$\begin{aligned} R1 (2 + 7 + 4 + 1) & \quad * 35 = 490 \\ R4 (2 + 7 + 4 + 1 + 1) & \quad * 30 = 450 \\ R3 (2 + 7 + 4 + 1 + 1 + 1) & \quad * 20 = 320 \\ R2 (2 + 7 + 4 + 1 + 1 + 1 + 1) & \quad * 15 = 255 \\ \hline \text{Total Run Time} & \quad 1515 \end{aligned}$$

TABLE 18. CALCULATIONS FOR INTERRUPT RULE MASK TECHNIQUE

$f_j / \Sigma f_j$	2,7	2,5	2,2	2,3			
$\Sigma f_j$	13	6	9	13			
$f_j$	35	15	20	30			
	R1	R2	R3	R4	$t_i$	$\Sigma f_j$	$\Sigma f_j / t_i$
C1	Y	N	Y	N	2	100	50
C2	N	-	Y	N	7	85	12.1
C3	Y	Y	-	N	4	80	20

Note: Assume  $t = 1$  for multiplying the data vector by the masking vector and compare the result with the transaction vector.

There are several strategies that can be used to decrease the run time. These strategies usually yield different results, and the one that produces a testing sequence with the lowest total run time should be selected for use in generating the translated code. The strategies do not guarantee optimal testing sequences in all cases, but they do show an improvement in minimizing object program run time (King [55]).

**STRATEGY A** tests the conditions in descending order of magnitude of relevance frequency ( $\Sigma f_j$ ). This is based on the supposition that it may be best to evaluate first those conditions most likely to be pertinent. With this strategy, the relative times of evaluation of the conditions are ignored. The testing sequence for Table 18 using this strategy would be C1-C2-R3-C3-R1-R4-R2:

$$\begin{aligned} R3 (2 + 7 + 1) & \quad * 20 = 200 \\ R1 (2 + 7 + 1 + 4 + 1) & \quad * 35 = 525 \\ R4 (2 + 7 + 1 + 4 + 1 + 1) & \quad * 30 = 480 \\ R2 (2 + 7 + 1 + 4 + 1 + 1 + 1) & \quad * 15 = 255 \end{aligned}$$

Total Run Time 1460

**STRATEGY B** tests the conditions in descending order of  $\Sigma f_j / t_i$ . This is based on the supposition that it may be best to evaluate first those conditions with the shortest evaluation times even though they may be less likely to be pertinent. This results in a testing sequence of C1-C3-R2-C2-R1-R4-R3:

$$\begin{aligned} R2 (2 + 4 + 1) & \quad * 15 = 105 \\ R1 (2 + 4 + 1 + 7 + 1) & \quad * 35 = 525 \\ R4 (2 + 4 + 1 + 7 + 1 + 1) & \quad * 30 = 480 \\ R3 (2 + 4 + 1 + 7 + 1 + 1 + 1) & \quad * 20 = 340 \end{aligned}$$

Total Run Time 1450

**STRATEGY C** tests the rules in descending order of frequency, evaluating conditions only when they become necessary for testing the rule. The testing sequence of Table 18 using this strategy would be C1-C2-C3-R1-R4-R3-R2:

$$\begin{aligned} R1 (2 + 7 + 4 + 1) & \quad * 35 = 490 \\ R4 (2 + 7 + 4 + 1 + 1) & \quad * 30 = 450 \\ R3 (2 + 7 + 4 + 1 + 1 + 1) & \quad * 20 = 320 \\ R2 (2 + 7 + 4 + 1 + 1 + 1 + 1) & \quad * 15 = 255 \end{aligned}$$

Total Run Time 1515

STRATEGY D tests the rules in descending order  $f_i/\Sigma t_i$ . This is based on the supposition that it may be best to test first those rules with relatively shorter condition evaluation times, even though they may have lower frequencies. This results in a testing sequence of C1-C2-C3-R1-R4-R3-R2 for Table 18 and a total run time of 1515.

For Table 18, the best testing sequence is derived by using STRATEGY B, which yields a total test time of 1450. This strategy would then be used to translate the decision table into a machine executable sequence of instructions.

The interrupt rule mask technique will use more storage than the simple rule mask technique because of greater program complication. The algorithm relies on user-supplied condition testing times and rule frequency of occurrence. These disadvantages can be outweighed by the marked savings in run time, so users that have large tables should consider using this method rather than the simple rule mask technique.

#### Conditional Testing and Network Techniques (Tree Structure Techniques)

The basis for the more sophisticated network or tree techniques are two algorithms, due to Press [89], for evaluating nonambiguous limited entry and extended entry decision tables.

In the quick-rule method the objective is to make, as soon as possible, those tests which will isolate a rule. This technique reduces the amount of storage required because it minimizes the number of branching instructions.

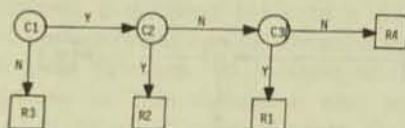
In Table 19 the condition portion of a decision table is shown. On the right of the table is the row count matrix which indicates the number of occurrences of each value in the condition entries of each row. For example, in the first row there are three "1's" and one "0". It can be seen that the smallest nonzero number in the row count matrix is in row one so the conditional interrogations associated with this row would be made. This would isolate rule 3 as indicated in the flow diagram in Table 19. A new subtable and row count matrix are con-

TABLE 19. QUICK-RULE DECISION TABLE

	R1	R2	R3	R4	
C1	1	1	0	1	ROW COUNT
C2	0	1	1	0	1 0
C3	1	0	1	0	3 1
					2 2
					2 2

SUBTABLE 1

	R1	R2	R4	
C1	1	1	1	ROW COUNT
C2	0	1	0	1 0
C3	1	0	0	3 0
				1 2
				1 2



structed to test the remaining rules. Row two and row three both have the same smallest occurrence value, so they are interrogated, and this isolates the remaining rules. The flow diagram in Table 19 depicts the final result of the quick-rule method as applied to the table.

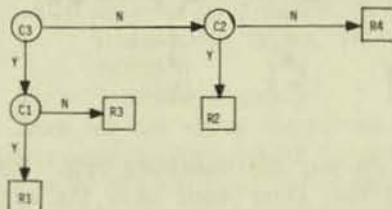
The objective of the delayed-rule method is to delay the tests which isolate rules as long as possible. This results in minimizing the average number of executed instructions (Montalbano [71]). An example of the delayed-rule method is shown in Table 20. Here, the row count matrix is searched for a conditional interrogation which will divide the table into two subtables as equal in size as possible. In Table 20 the original table is divided evenly into two subtables and their respective row count matrices. The flow diagram indicates the testing sequence of the conditional rows with respect to minimum row count occurrences in the subtables. Comparing Tables 19 and 20, it can be observed that fewer instructions will be required to isolate a rule using the delayed-rule method. The delayed-rule method minimizes the average number of executed instructions, so it will have less run time than the quick-rule method.

The foregoing network type algorithms can develop greater efficiency in translating a decision table into a computer program

TABLE 20. DELAYED-RULE DECISION TABLE

	R1	R2	R3	R4	ROW COUNT
C1	1	1	0	1	3
C2	0	1	0	0	1
C3	1	0	1	0	2

SUBTABLE 1		ROW COUNT	SUBTABLE 2		ROW COUNT
	R1	R3		R2	R4
C1	1	0	1	1	1
C2	0	0	0	2	0
C3	1	1	2	0	0



by using a more complex algorithm. If several pre-known conditions, such as rule frequency, dash count, delta count, and weighted dash count are available, more efficient algorithms can be used for minimizing core storage, and minimizing run time. Several terms needed to be defined before discussing the algorithms.

The *Column Count* (CC) for a rule is equal to  $2^r$ , where  $r$  is the number of dashes (don't-care entries) in the rule. The *Delta Count* (Delta) for a row is the absolute value of the difference between the number of Y's in a row and number of N's in that row. In each row, the *Dash Count* (DC) is equal to the sum of the column counts of all rules that have a dash entry in the row. A *Weighted Dash Count* (WDC) for a row is equal to the sum of the products of rule frequencies and column counts of all rules that have a dash entry in the row.

The testing sequence for both algorithms is (Pollack [85]):

- 1) One row of the original decision table is selected—the criterion for selection differs for the two algorithms.
- 2) The original decision table is then de-

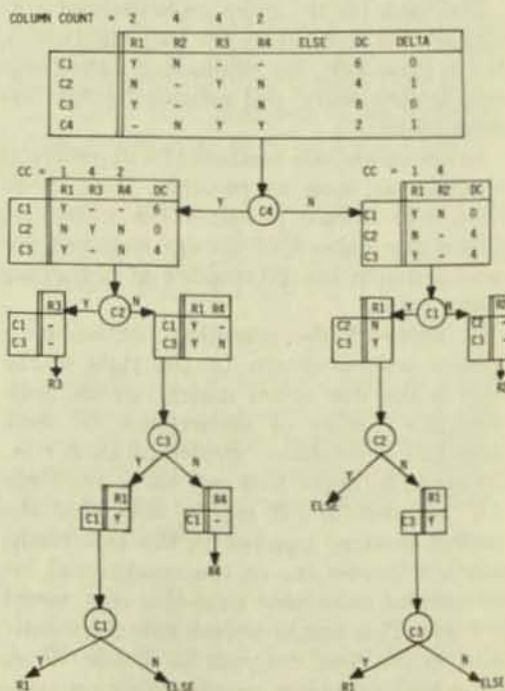
composed into two subtables (containing one less row), one subtable and one rule, or two rules; each of these is associated with each branch of the comparison.

- 3) A row is then selected from each of these subtables, and a condition becomes attached to the previously selected condition; i.e., a single condition row is selected and becomes the next comparison of the testing sequence.
- 4) The process is continued until each rule of the original decision table or an ELSE rule appears as one of the branches of the condition, or a subtable indicates that the original table contained redundant or contradictory rules.

#### Quick-Rule Algorithm

The objective of the quick-rule algorithm to be discussed is to minimize the number of storage locations (Pollack [85]). This procedure is illustrated in Table 21, and the

TABLE 21. MINIMUM CORE STORAGE



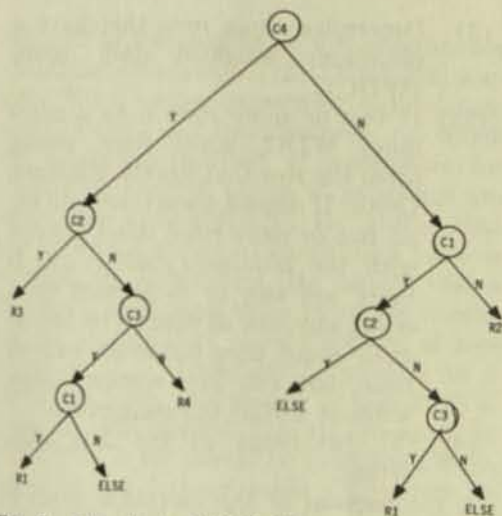


Fig. 8. Flowchart of Table 21.

resulting test sequence shown in Figure 8. The steps in the algorithm are:

- (1) Check the table for redundancies and contradictions. If two rules do not contain at least one row where one rule has a Y entry and the other has an N entry, the two rules are either redundant or contradictory: they are redundant if they have the same action and contradictory if they do not.
- (2) Calculate the column count (CC) and dash count (DC).
- (3) Determine the row that has the minimum dash count. If two or more rows have the minimum dash count, select the row that has the maximum Delta.
- (4) Taking the row selected in (3), use the YES-NO branch to create two subtables, each containing one or more rules, with one row less than the original row.
- (5) If the subtable contains more than one rule return to (1).
- (6a) If the subtable has exactly one rule that contains only dashes, that rule has been isolated.
- (6b) If the subtable has exactly one rule that contains only dashes, choose any non-dash row and discriminate on it. This will yield a subtable from the satisfied condition and an

ELSE rule isolation from the opposing branch.

(6c) If no subtable is produced, an ELSE rule isolation is indicated.

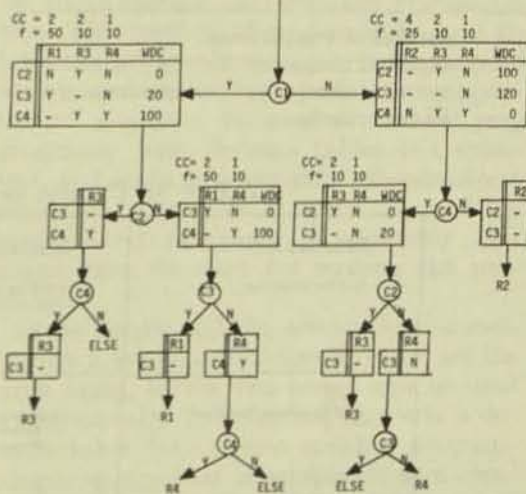
(6d) If the subtable has exactly one rule that has one condition with a Y or N entry, discriminate on the condition. The satisfied branch isolates the rule, while the opposing branch isolates the ELSE rule.

*Delayed-Rule Algorithm*

The objective of the second algorithm is to convert a decision table to a program whose comparisons can be executed in minimum time (Pollack [85]). Some of the assumptions in this algorithm are: a) any rules not specified or implied in the table are assumed to be part of an ELSE rule; b) systems analysts can provide estimates of how often each rule in the table will be satisfied by an average batch of transactions to be tested; and c) relatively few transactions will satisfy the ELSE rule. Table 22 illustrates the procedure for the sec-

TABLE 22. MINIMUM RUN TIME

	R1	R2	R3	R4	ELSE	MDC
C1	Y	N	-	-	-	60
C2	N	-	Y	N	-	100
C3	Y	-	-	N	-	140
C4	-	N	Y	Y	-	100





ond algorithm, and Figure 9 shows the resulting test sequence. The algorithm's procedure follows:

(1), (2) Same as the previous algorithm.

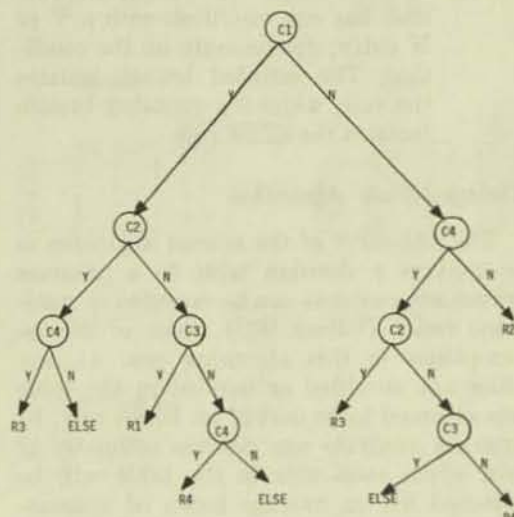


FIG. 9. Flowchart of Table 22.

TABLE 23. COMPARISON OF TABLES 21 AND 22

RULE NUMBER	TABLE 21		TABLE 22	
	a	b	a	b
1	$4 \times 50 = 200$		$3 \times 50 = 150$	
2	$2 \times 25 = 50$		$2 \times 25 = 50$	
3	$2 \times 10 = 20$		$3 \times 10 = 30$	
4	$3 \times 10 = 30$		$4 \times 10 = 40$	
ELSE	$(3.6_c) \times 5 = 18$		$(3.6_c) \times 5 = 18$	
	318		288	

a = number of comparisons.

b = expected frequency.

c = average number of comparisons for 3 ELSE branches.

(3) Determine those rows that have a minimum weighted dash count (WDC).

(3a) If two or more rows have a minimum WDC, select from among them the row that has the minimum Delta. If among these there still exist two or more rows, select the row with the minimum dash count. If there are two or more such rows, select any one of them. The test on dash count does not affect running time, but can save memory space without adding to running time.

(4), (5), (6) Same as the first algorithm.

A comparison of the execution times of the two algorithms depicted in Tables 21 and 22 is shown in Table 23. The execution time is based on processing 100 transactions which have a frequency distribution as indicated in Table 22. Assuming each conditional interrogation takes one time unit, the total test times are indicated in Table 23. It can be seen that the second algorithm is more efficient in run time because it uses 288 time units to process the 100 transactions, whereas the first algorithm uses 318 time units.

A summary of the various decomposition and conversion algorithms is given in Table 24. This table contains both the short (common) and the long class name, as well as major references.

#### Ambiguities

An aspect that is often ignored in decision table processing is that of ambiguities in the tables. These need to be somehow

TABLE 24. CLASSES OF CONVERSION ALGORITHMS

Short Class Name	Long Class Name	Sub-Algorithms
Masking Technique	Scanning and Rule Mask Technique	Rule Mask Algorithm Kirk (82)
		Interrupt Rule Mask Algorithm King (55)
Tree Structure Technique	Conditional Testing and Network Techniques Press (89)	Quick-Rule Algorithm Montalbano (71), Pollack (85)
		Delayed Rule Algorithm Montalbano (71), Pollack (85) Shwayder (98)

reported to the analyst or the programmer (Muthukrishnan, et al. [73]). King [58] suggests that the rules concerning redundancy, contradiction, and completeness, on which the diagnostic facilities of processors for translating decision tables to programs are based, are unsatisfactory. He states that the important aspect of checking a table is to eliminate ambiguities. He asserts that a check-out of decision table input (in checking for ambiguities) should consist of two parts: 1) if no ambiguity is possible in a particular table, this should be noted; 2) if there are ambiguities then all outcomes in which they occur should be produced, leaving it to the decision table originators to check these facets of the table.

King, in a later paper [61] presents a slightly different approach than the one-rule convention to ambiguities involving multi-rule decision tables. This approach retains the idea that the set of actions corresponding to only one rule is selected for a particular transaction. However, it does not insure it by allowing only one rule to be satisfied, but permits two or more rules to be satisfied provided they have the same action entries. Accordingly, if more than one rule can be satisfied by a transaction with identical action entries, the ambiguity is said to be "apparent," whereas if transactions specify different action entries, the ambiguity is said to be "real," requiring correction (King [61]).

Execution time diagnostics for these ambiguities, as opposed to compile time diagnostics, are implemented by condition tests that provide complete information about the conditions and their relation to the data. The tree methods can not really cope with real ambiguities, and the rule mask techniques do not consider them. Execution time diagnostics for these ambiguities enhance the value of decision table usage in programming (Muthukrishnan, et al. [73]).

One of the latest techniques developed is that of Muthukrishnan and Rajaraman [73], which uses execution time diagnostics in pinpointing ambiguities in decision tables. They contend that execution time diagnostics are of immense value in checking out decision tables, because they precisely

pinpoint the errors in logic, instead of passing the task to the systems analyst. Pollack [87] and King [61] have disputed this idea and state that it is better to find ambiguities during compilation rather than during execution. In their work, Muthukrishnan and Rajaraman developed two algorithms for programming decision tables, which have the merits of simplicity of implementation, and detection of ambiguities at execution time. The first algorithm is for limited entry decision tables, and clarifies the importance of proper coding in simplifying the mechanics of rule matching; the second algorithm programs a mixed entry decision table directly, without any intermediate conversion, to a limited entry form, which results in storage economy.

#### Automatic Versus Manual Translation

Effective programming efforts are required to convert decision tables into operational computer programs. This conversion means that tabular representation of information and data must be converted into machine language instructions. The techniques of program conversion have been well developed in the past few years. There are four principal approaches which can be used: *manual*, *semiautomatic*, *interpretation*, and *automatic conversions* (Glans, et al. [39]).

*Manual processing* is accomplished by programmer rewriting of each decision table for more efficient and compact representation. This method offers flexibility, and allows the programmer to take advantage of testing certain rules or conditions in a particular sequence. In general, manual programming from decision tables is convenient, and leads to reasonably efficient object programs (Glans, et al. [39]). Any of the higher level languages or assembly language may be used for writing the programs.

Some people call the *semiautomatic conversion* a translator. Basically they are the same thing, so the two terms will be used synonymously. This method converts a decision table format into another programming language that is acceptable as a computer input language. One advantage of

processing a table in this manner is that it can be converted into a language such as FORTRAN or COBOL. Thus a table can subsequently be run on any machine that accepts FORTRAN or COBOL. One disadvantage of this method is its relative inefficiency. It requires a two-step process, because the decision table has to be translated into a programming language, and then this source language has to be compiled or assembled into an object program.

The *interpretive conversion* allows for direct storage of the decision table, usually in a coded or compact form, thereby insuring easier maintenance (McDaniel [67]). It is necessary to have the interpretive program in core before inputting the source program. The main disadvantage is the slower solution speed. These programs usually have some restrictions on the type of format and vocabulary used in the source program. While this method lacks the sophistication of the other conversion methods, it offers easier program maintenance.

*Automatic conversion* programs are those which will accept decision tables written in a user source language, and completely convert them to a fully acceptable input, usually at the machine language level (McDaniel [67]). Generally, an automatically converted decision table will require less execution time than interpretive and semi-automatic conversions. This method forces a higher degree of standardization, thus it may encourage more effective communication. The disadvantages of automatic conversion are that it tends to be inflexible and is computer-oriented. Conversion of such a processor from one machine to another would require a considerable amount of reprogramming, unless, of course, the processor is written in a higher level language, such as COBOL; e.g., DETAP (Pollack, et al. [86]).

Before selecting the method of conversion, it is desirable to analyze the methods previously discussed, and then to select the one that best fits the situation. Some questions that should be asked during the evaluation are: (1) Is it possible to use the method of conversion? (2) What are the restrictions of the possible methods of conversion? (3)

Does the processor produce an efficient code that satisfies the requirements? (4) With respect to the processors that satisfy (3), is the cost of running the processor worth the service it provides (Gildersleeve [34])? After answering these questions and comparing the different conversion methods, it may be found that the most economical solution is to hand code the programs from decision tables, and not to use a preprocessor at all.

Adding, deleting, and restructuring tables is comparable to developing original tables and programs. If a change is to be made in a table, usually extensive hand conversion is needed, again adding more overhead to the desired change. (These overhead costs must be included when considering long-term maintenance.)

#### IV. CONCLUSION

Decision tables can be a powerful aid in programming, documentation, and in effective man-to-man and man-to-machine communications. Inherent in the design of a decision table is the visual presentation of complex programming logic with relative ease for modification, implementation, and automatic conversion into executable programs. Several such algorithms for converting decision tables to programs, by either manual or automatic techniques, were shown to be feasible, as well as practical for implementation.

An inevitable outcome of increasing use of decision tables in programming has been the development of a large number of package processors and translators for the conversion of decision tables to functional program form. These decision table processors are software programs, which are available for almost any language and hardware configuration. Each processor has standards as to size, format, words in the statement portion, and other required characteristics which must be met by tables prior to processing. Meeting these standards will usually require some manual checking of the tables prior to the preparation of the computer input. Then, the processor may reveal redundancies, missing situations, and

contradictions within the table (McDaniel [68]).

Since each table generates a separate segment of coding, each segment can be traced back to the table that generated it; therefore, changes can be easily made by reworking one or more of the tables, and the effect of such changes observed.

Each processor generates straightforward coding, free of programming tricks; thus a programmer should be able to follow any program in a given installation, and make any necessary changes.

A detailed reference table is given in McDaniel's "Decision Table Software" [68], which examines characteristics of many of the processors, including the language of the processor and the output language; the hardware for which a processor has been implemented (in some cases the hardware for which it is being developed); the types of tables accepted as input by a given processor; the cost and availability of a given processor; the number of tables, rules, conditions, and actions allowed by a given processor; and other notable characteristics.

The algorithm and translator used in converting a decision table into a computer program will therefore be determined by the extent of the processing facilities and the constraints of the application program. To be able to use different algorithms and translators provides more flexibility for the users and yet works against the popularity of decision tables in programming because more individual effort is required in determining which algorithm and translation process to use.

#### ACKNOWLEDGMENT

The author is deeply indebted to the referees for their valuable suggestions and, in particular, to the referee who pointed out a major omission in the original manuscript and provided encouragement in overcoming this difficulty.

#### BIBLIOGRAPHY

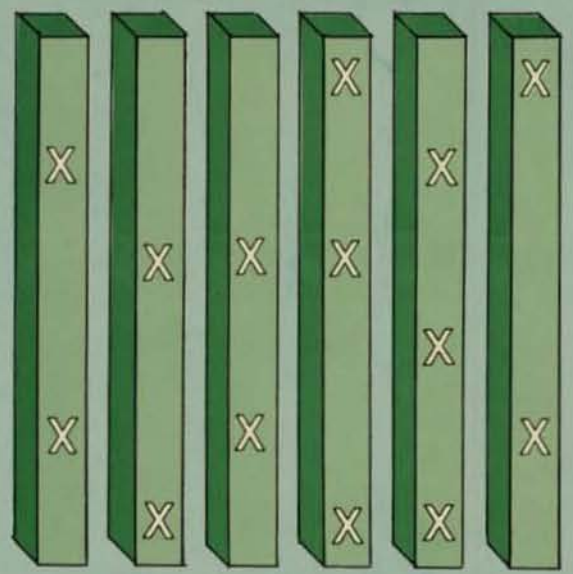
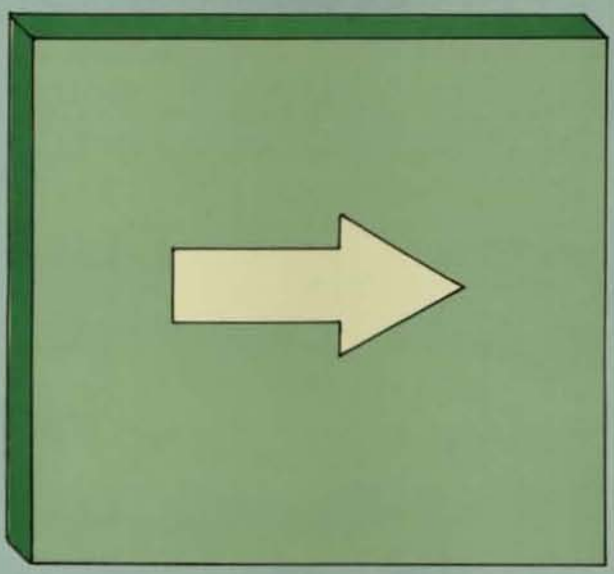
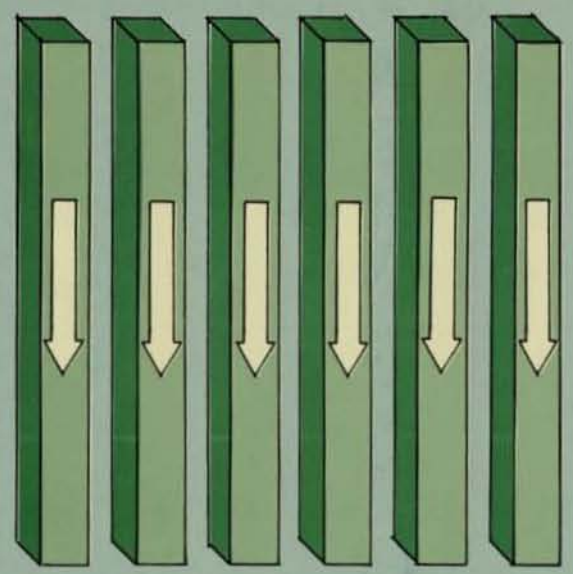
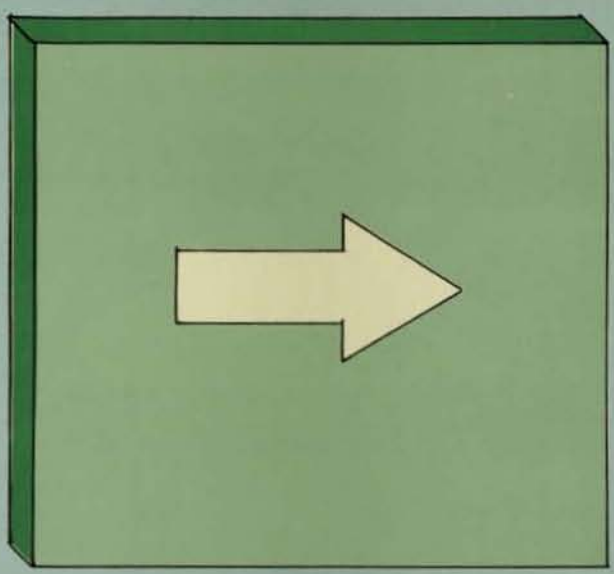
1. AINSLIE, R. J., AND KENNEY, A. A. "A tool for for tax practitioners." *The Tax Advisor* (June 1972), 336-345.

2. ARNOLD, H. O. "Utilization of a decision table translator for basic program creation." *SIGPLAN Notices*, 6, 8 (Sept. 1971), 12-19.
3. AUERBACH. "Decision tables—their general construction and acceptance in programming." *Auerbach Standard EDP Reports* (1968), pp. 23:030:100-103.
4. ARMERDING, G. W. "FORTAB: a decision table language for scientific computing applications." Rand Corporation, **RM-3306-PR** (Sept. 1962), p. 39.
5. BARNARD, T. J. "A new rule mask technique for interpreting decision tables." *The Computer Bulletin*, Vol. 13 (May 1969), 153.
6. BJORK, HARRY. "Decision tables in ALGOL 60." *BIT*, 8(1968), 147-153.
7. CALKINS, L. W. "Place of decision tables and DETAB-X." *Proceedings Decision Tables Symposium* (Sept. 1962), 9-12.
8. CANNING, R. G. "How to use decision tables." *EDP Analyzer* 4, 5 (May 1966).
9. CANTRELL, N. H., KING, J., AND KING, F. G. H. "Logic structure tables." *Comm. ACM* 4, 6 (June 1961), 272-275.
10. CANTRELL, N. H. "Commercial and engineering applications of decision tables." *Proceedings Decision Tables Symposium* (Sept. 1962), 55-61.
11. CHAPIN, NED. "A Guide to decision table utilization." *Data Processing Proceedings 1966*, Vol. 11 (1966), 327-329.
12. CHAPIN, NED. "Parsing of decision tables." *Comm. ACM* 10, 8 (August 1967), 507-512.
13. CHAPIN, NED. "An introduction to decision tables." *DPMA Quarterly* 3, 3 (April 1967), 3-23.
14. CHAPIN, NED. *Flowcharts*. Auerbach Publishers, Princeton, N.J., 1971, pp. 20-21.
15. CHAPMAN, A. E., AND CALLAHAN, M. A. "A description of the basic algorithm used in the DETAB/65 Preprocessor." *Comm. ACM*, 10, 7 (July 1967), 447-446.
16. CODASYL Systems Group and Joint Users of ACM. *Proceedings Decision Tables Symposium* (Sept. 1962).
17. CODASYL Systems Development Group. "Decision tables tutorial using DETAB/X" (1962).
18. CODASYL Systems Group, DETAB-X. "Preliminary specifications for a decision tables structured language" (1962).
19. CODASYL, Decision Table Task Force of Systems Committee. "Draft of decision table standards" (March 1966).
20. DENOLF, H. "Decision tables: an annotated bibliography." *IAG Quarterly* 1 (1968), 67-82.
21. DEVINE, D. J. "LOBOC, logical business oriented coding." Insurance Company of North America, Oct. 1962.
22. DEVINE, D. J. "Decision tables as a basis of a programming language." *DPMA Quarterly* 7 (1965), 461-466.
23. DIXON, P. "Decision tables and their application." *Computers and Automation* 13, 4 (April 1964), 14-19.

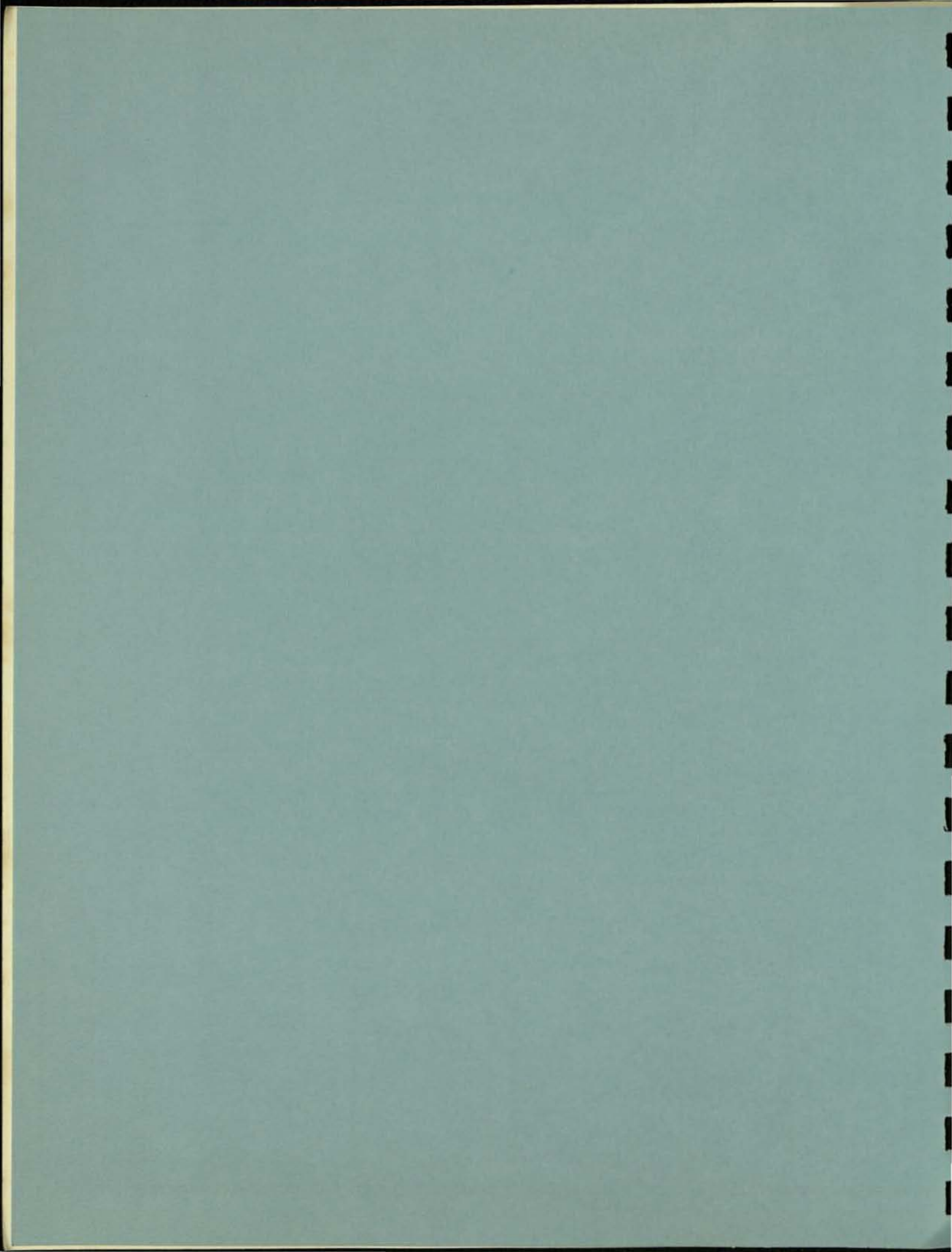
24. EGLER, J. F. "A Procedure for converting logic table conditions into an efficient sequence of test instruction." *Comm. ACM* **6**, 8 (Sept. 1963), 510-514.
25. ELLIS, J. "Decision tables, a users' guide." Western Electric Company, June 1967.
26. EVANS, O. Y. "Reference manual for decision tables." IBM, Sept. 1961.
27. FERGUS, R. M. "Decision tables—an application analyst/programmer's view." *Data Processing* **12** (1967), 85-109.
28. FERGUS, R. M. "An introduction to decision tables." *Systems and Procedures Journal* (July-August 1968), 24-27.
29. FERGUS, R. M. "Good decision tables and their uses." *Systems and Procedures Journal* (Sept.-Oct. 1968), 18-21.
30. FIFE, R. C. "Decision tables, UNIVAC application report." *Spring Joint Computer Conference of Systems and Procedures Association* (April 1965).
31. FIFE, R. C. "Decision tables." Systems Programming Dept., UNIVAC, 1966.
32. FISHER, D. L. "Data Documentation and Decision Tables." *Comm. ACM* **9**, 1 (Jan. 1966), 26-31.
33. FLETCHER, G. R. "Seminar on decision tables." Bureau of the Census, Sept. 1969.
34. GILDERSLEEVE, T. R. *Decision Tables and Their Practical Application in Data Processing*. Prentice-Hall, Englewood Cliffs, N.J., 1970.
35. GENERAL ELECTRIC COMPANY. "GE-225 TABSOL reference manual and GE-224 TABSOL application manual." CPB-147B, June 1962.
36. GLANS, R. B., AND GRAD, B. "Tabular descriptive language." *IBM Technical Report 245* (Jan. 1962).
37. GLANS, R. B., AND GRAD, B. "7080 decision table system preliminary manual." *IBM Technical Report 2D1* (April 1962).
38. GRAD, B. "Structure and concept of decision tables." *Proceedings Decision Tables Symposium* (Sept. 1962), 19-28.
39. GRAD, B. "Engineering data processing using decision tables." *Data Processing* **8** (1965), 467-476.
40. GRINDLEY, C. B. B. "The use of decision tables within systematics." *The Computer Journal* **11**, 2 (August 1968), 128-133.
41. GRINDLEY, C. B. B. "Systematics—a non-programming language for designing and specifying commercial systems for computers." *The Computer Journal* **9**, 2 (August 1966), 124-128.
42. HARRISON, W. J. "Practically complete decision tables: a range approach." *SIGPLAN Notices* **6**, 8 (Sept. 1971), 89-93.
43. HAWES, M. K. "The need for precise definition." *Proceedings Decision Tables Symposium* (Sept. 1962), 13-18, 20-21.
44. HAWES, M. K. "The use of decision tables for problem specifications." *Proceedings UNIVAC Users Association* (April 1965), 55-61.
45. HIRSCHHORN, E. "Simplification of a class of Boolean functions." *J. ACM* **5**, 1 (Jan. 1958), 67-75.
46. HONEYWELL, INC. *An Introduction to Decision Tables—A Programmed Text*, 1st Ed., Oct. 1969.
47. HUGHES, M. L., SHANK, R. M., AND STEIN, E. S. "Decision tables." MDI Publications, 1968.
48. IBM CORPORATION. "1401 decision logic translator H20-0063," and "1401 decision logic translator H20-04921." *Decision Tables—Practice Problems and Solutions*, 1963.
49. IBM CORPORATION. "Decision tables—a systems design and documentation technique." *IBM*, **F20-8102** (1962), p. 21.
50. KAVANAGH, R. F. "TABSOL—a fundamental concept for systems oriented languages." *Eastern Joint Computer Conference*, Vol. 18 (Dec. 1960), 13-15, 117-136.
51. KAVANAGH, R. F. "TABSOL—the language of decision making." *Computers and Automation* **10**, 9 (Sept. 1961), 15, 18-22.
52. KAVANAGH, R. F., AND ALLEN, M. "The use of decision tables." *Proceedings 1963 Conference of International DPMA*, 318.
53. KAVANAGH, R. F., AND SCHMIDT, D. T. "Using decision structure tables, Part I: Principles and preparation; Part II: Manufacturing application." *Datamation*, Vol. 10, (Feb.-March, 1964).
54. KING, J. E. "LOGTAB: a logic table technique." *General Electric March* 1959.
55. KING, P. J. H. "Conversion of decision tables to computer programs by rule mask techniques." *Comm. ACM* **9**, 11 (Nov. 1966), 796-801.
56. KING, P. J. H. "Some comments on systematics." *The Computer Journal* **10**, 1 (May 1967) 116-119.
57. KING, P. J. H. "Decision tables." *The Computer Journal* **10**, 2 (August 1967), 135-142.
58. KING, P. J. H. "Ambiguity in limited entry decision tables." *Comm. ACM* **11**, 10 (Oct. 1968), 680-684.
59. KING, P. J. H. "The interpretation of limited entry decision table format and relationships among conditions." *The Computer Journal* **12**, 4 (Nov. 1969), 320-326.
60. KING, P. J. H., AND JOHNSON, R. G. "Some comments on the use of ambiguous decision tables and their conversion to computer programs." *Comm. ACM* **16**, 5 (May 1973), 287-290.
61. KIRK, G. W. "Use of decision tables in computer programming." *Comm. ACM* **8**, 1 (Jan. 1965), 41-43.
62. KLICK, D. C. "TABSOL." Preprints of Summaries of Paper presented at National ACM, Paper 10, B-2 (Sept. 1961).
63. LOMBARDI, L. A. "A general business-oriented language based on decision expressions." *Comm. ACM* **7**, 2 (Feb. 1964), 104-111.
64. LONDON, K. *Decision Tables: A Practical Approach for Data Processing*. Auerbach Publishers, Princeton, N.J., 1972.
65. LUDWIG, H. R. "Simulation with decision

- tables." *Journal of Data Management* 6 (Jan. 1968), 20-27.
67. McDANIEL, H. *Applications of Decision Tables*. Brandon/Systems Press, Princeton, N.J., 1970.
  68. McDANIEL, H. *Decision Table Software*. Brandon/Systems Press, Princeton, N.J., 1970.
  69. McDANIEL, H. *An Introduction to Decision Logic Tables*. John Wiley, New York, 1968.
  70. MEYER, H. I. "Decision tables as an extension to programming languages." *Data Processing* 8 (1965), 477-483.
  71. MONTALBANO, M. "Tables, flowcharts and program logic." *IBM Systems Journal* (Sept. 1962), 51-63.
  72. MORGAN, J. J. "Decision tables." *Management Services* (Jan.-Feb. 1965), 13-18.
  73. MUTHUKRISHNAN, C. R., AND RAJARAMAN, V. "On the conversion of decision tables to computer programs." *Comm. ACM* 13, 6 (June 1970), 247-251.
  74. NARAMORE, F. "Application of decision tables to management information systems." *Proceedings Decision Tables Symposium* (Sept. 1962), 63-74.
  75. NICKERSON, R. C. "An engineering application of logic structure tables." *Comm. ACM* 4, 11 (Nov. 1961), 516-520.
  76. PEEL, ROGER. "Decision table translation." *The Computer Bulletin* 13, 12 (Dec. 1969).
  77. POLLACK, S. L. "What is DETAB-X?" *Proceedings Decision Tables Symposium* (Sept. 1962).
  78. POLLACK, S. L. "DETAB-X: an improved business-oriented computer language." Rand Corporation Memo RM-3273-PR (August 1962).
  79. POLLACK, S. L., AND WRIGHT, K. R. "Data description for DETAB-X." Rand Corporation Memo RM-3010-PR (March 1962).
  80. POLLACK, S. L. "Analysis of the decision rules in decision tables." Rand Corporation Memo RM-3669-PR (May 1963).
  81. POLLACK, S. L. "How to build and analyze decision tables." Rand Corporation Memo P-2829 (Nov. 1963).
  82. POLLACK, S. L. "CODASYL, COBOL, and DETAB-X." *Datamation* 9, 2 (Feb. 1963), 61.
  83. POLLACK, S. L. "The development and analysis of decision tables." *Ideas for Management*, 1964, International Systems Meeting, Systems and Procedures Association, Philadelphia, 1964.
  84. POLLACK, S. L. "Decision tables for systems design." *DPMA Quarterly* 8 (1965).
  85. POLLACK, S. L. "Conversion of limited entry decision tables to computer programs." *Comm. ACM* 8, 11 (Nov. 1965) 677-682.
  86. POLLACK, S. L., AND HARRISON, W. J. "DETAP version III user's guide." *IMI*, July 1969.
  87. POLLACK, S. L. "Comment on the conversion of decision tables to computer programs." *Comm. ACM* 14, 1 (Jan. 1971), 52.
  88. POLLACK, S. L., HICKS, H., AND HARRISON, W. J. *Decision Tables: Theory and Practice*. Wiley, New York, 1971.
  89. PRESS, L. J. "Conversion of decision tables to computer programs." *Commun. ACM* 8, 6 (June 1965), 385-390.
  90. QUINE, W. B. "The problem of simplifying truth functions." *American Math. Mon.*, Vol. 59 (1952), 521-531.
  91. QUINE, W. B. "A way to simplify truth functions." *American Math. Mon.*, Vol. 62 (1965), 627-631.
  92. REINWALD, L. T., AND SOLAND, R. M. "Conversion of limited entry decision tables to optimal computer programs. I: Minimum average processing time." *J. ACM* 13, 3 (July 1966), 339-358.
  93. REINWALD, L. T. "An introduction to TAB40." Research Analysis Corporation, Nov. 1966.
  94. REINWALD, L. T., AND SOLAND, R. M. "Conversion of limited entry decision tables to optimal computer programs. II: Minimum storage requirements." *J. ACM* 14, 4 (Oct. 1967), 742-758.
  95. ROBINSON, F. "Processing of decision tables in COBOL." *Computer Weekly* No. 222/223 (Dec. 1970).
  96. SHAW, C. J. "Decision tables—an annotated bibliography." *S. D. C.*, TM-2288/000/00, Dec. 1965.
  97. SHAW, C. J. (ED.) "Decision tables." *SIGPLAN Notices* 6, 8 (Sept. 1971), 1-111.
  98. SHWAYDER, K. "Conversion of limited entry decision tables to computer programs—a proposed modification to pollack's algorithm." *Comm. ACM* 14, 2 (Feb. 1971), 69-73.
  99. SLAGLE, J. R. "An efficient algorithm for finding certain minimum cost procedures for making binary decisions." *J. ACM* 11, (1964), pp. 253-264.
  100. SPRAGUE, V. G. "Letters to the Editor" (On Storage Space of Decision Tables). *Comm. ACM* 9, 5 (May 1966), 319.
  101. ST. CLAIR, P. R., JR. "Decision tables clear the way for sharp selection." *Computer Decisions* 12, 2 (Feb. 1970), 14-18.
  102. STRUNZ, H. "The development of decision tables via parsing of complex decision situations." *Comm. ACM* 16, 6 (June 1973), 366-369.
  103. TAYLOR, H. *Decision Table Technique for Computer Systems*. Hirschfeld Press, Philadelphia, Pa., 1968.
  104. VEINOTT, C. G. "Programming decision tables in FORTRAN, COBAL, or ALGOL." *Comm. ACM* 9, 1 (Jan. 1966), 31-35.
  105. VERHELST, M. "Procedures for finding optimal and near optimal test sequences for applying rule mask techniques in object programs derived from decision tables." *IAG Quarterly* 1, (1968), 47.
  106. VERHELST, M. "A Technique for constructing decision tables." *IAG Quarterly* 2, 1 (1969), 27.
  107. WILLIAMS W. K. "Decision structure tables." *NAA Bulletin*, No. 9 (1965), 58-62.
  108. WRIGHT, K. R. "Approaches to decision table processors." *Proceeding Decision Tables Symposium* (Sept. 1962) 41-44.

File  
Dec Juntas



decibole

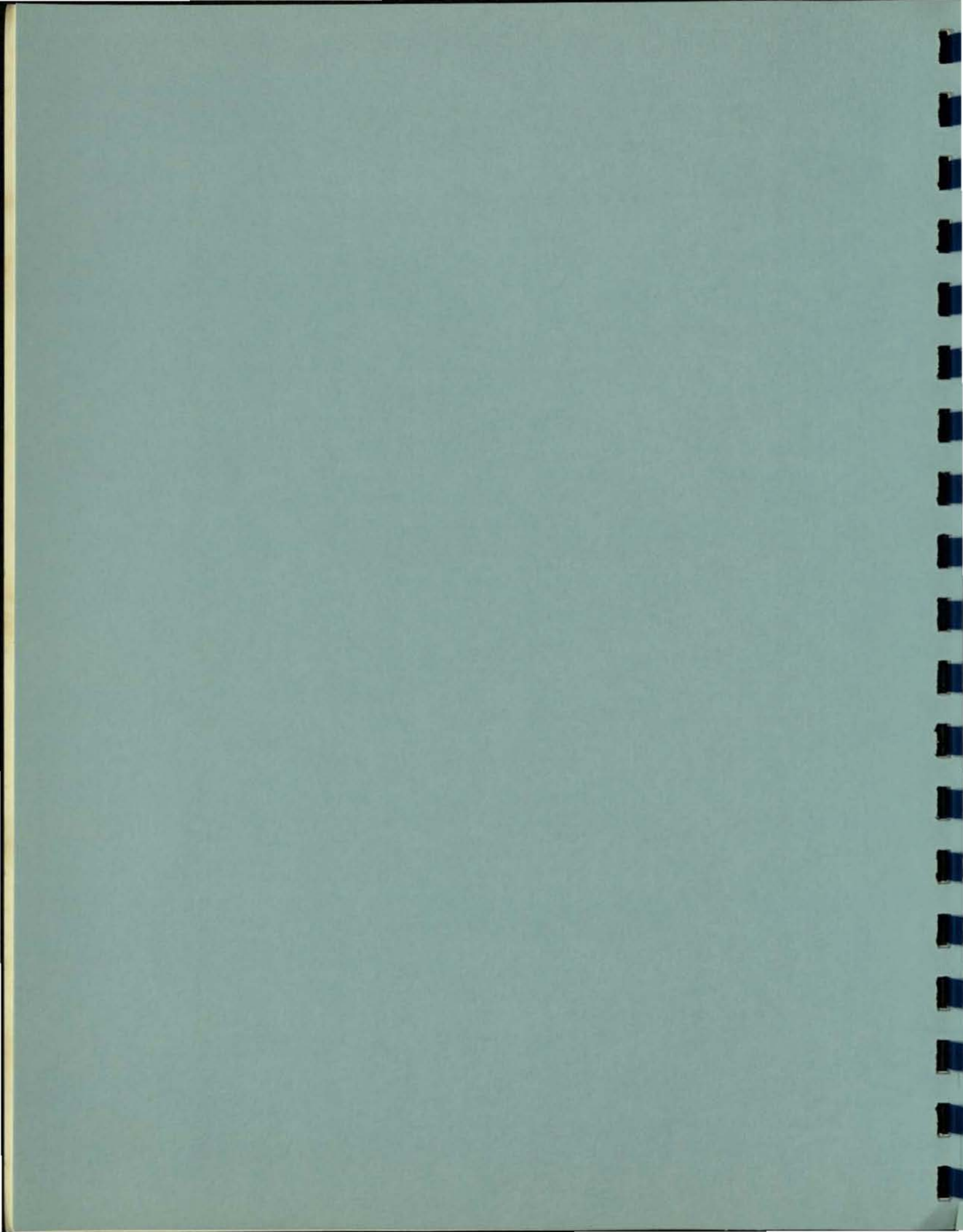




# TABLE OF CONTENTS

FOREWARD .....	3
WHAT IS DECIBLE III.....	3
LANGUAGE.....	3
HARDWARE REQUIREMENTS.....	3
TRAINING .....	3
THIS MANUAL .....	3
1. INTRODUCTION .....	5
USE OF DECISION TABLES .....	5
USE OF DECIBLE III .....	5
2. SYSTEM DESCRIPTION .....	7
GENERAL DESCRIPTION .....	7
INPUT TO DECIBLE III.....	7
OUTPUT OF DECIBLE.....	7
DECISION TABLE PROCESSING .....	8
SHORTHAND PROCESSING.....	8
3. SOURCE LANGUAGE LIBRARY MAINTENANCE SYSTEM .....	9
PURPOSE OF SYSTEM.....	9
GENERAL DESCRIPTION .....	9
SEQUENCE FIELD .....	9
INSERTING RECORDS .....	10
INSERTING A BLOCK OF RECORDS .....	11
REPLACING AND DELETING RECORDS.....	12
4. SHORT-HAND TRANSLATION SYSTEM .....	13
GENERAL DESCRIPTION .....	13
ABBREVIATION DEFINITION STATEMENT .....	13
EXAMPLES OF ABBREVIATION DEFINITIONS .....	14
5. WRITING DECIBLE III DECISION TABLES.....	15
GENERAL DESCRIPTION .....	15
EXTENDED AND MIXED ENTRY DECISION TABLES.....	15
CODING INSTRUCTIONS .....	15
DECISION TABLE IDENTIFICATION .....	17
CONDITION STATEMENTS .....	17
ACTION STATEMENTS .....	17
INITIAL SET ACTIONS .....	17
DECIBLE III SPECIAL ACTION STATEMENTS.....	18
ELSE RULE .....	18
END OF DECISION TABLE.....	19
LIMITED/EXTENDED ENTRY COMPARISON .....	19

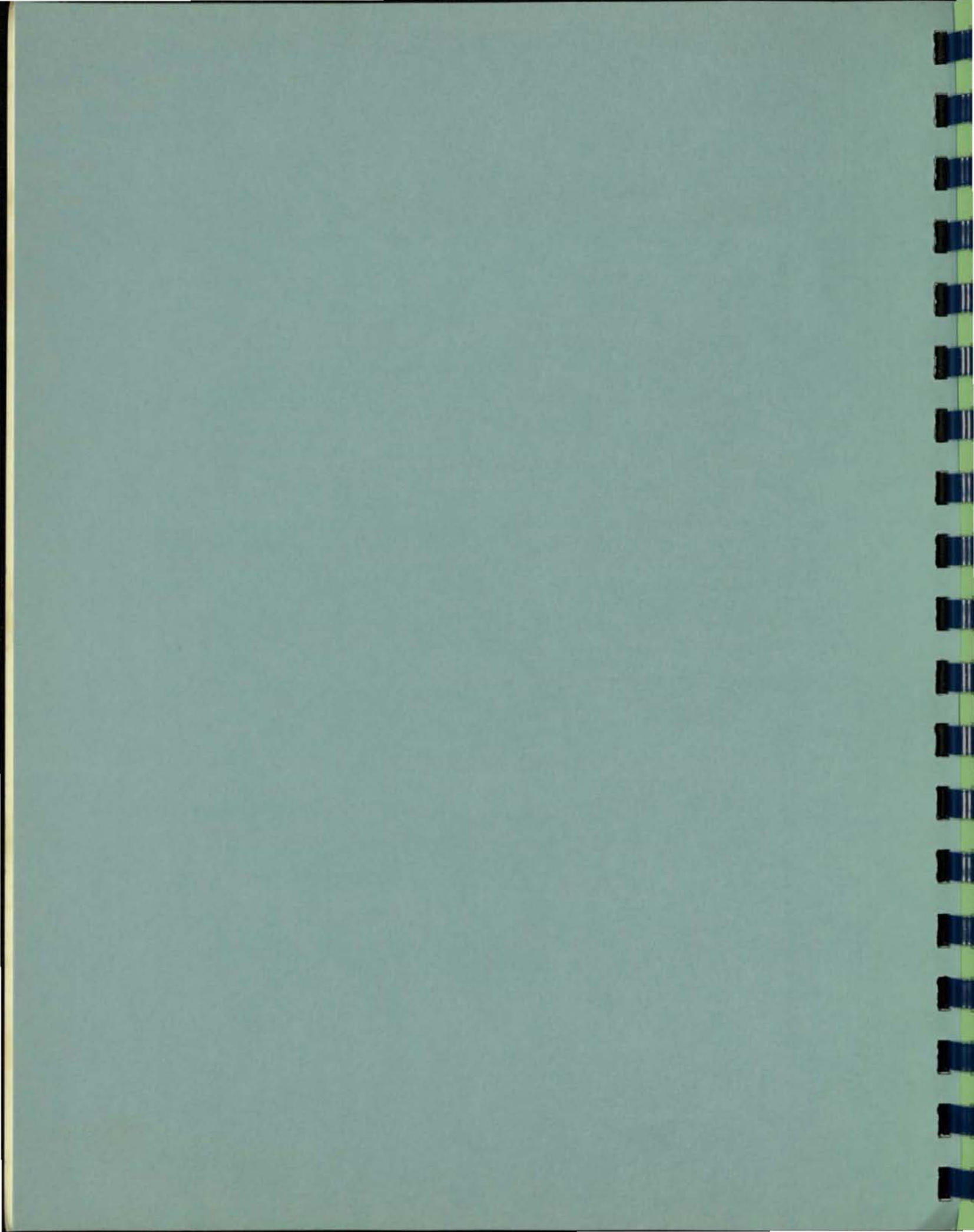




# TABLE OF CONTENTS

6. DECIBLE III STATEMENTS .....	21
GENERAL DESCRIPTION .....	21
DECIBLE OPTION STATEMENT .....	21
DECIBLE TABLE STATEMENT .....	24
X STATEMENT.....	26
DECIBLE SET STATEMENT .....	27
DECIBLE COMMENT STATEMENT .....	28
ABBREVIATION DEFINITION STATEMENT .....	29
7. PROGRAMMING GUIDELINES.....	31
TABLE ORDER .....	31
TABLE UNIQUENESS.....	31
VALUE RANGES .....	31
CONDITION STATEMENTS .....	32
ACTION STATEMENTS .....	32
RULES.....	32
APPENDIX A .....	A-1
DIAGNOSTICS .....	A-1
APPENDIX B – LIMITED ENTRY EXAMPLE .....	B-1
INPUT LISTING.....	B-1
OUTPUT LISTING.....	B-3
APPENDIX C – EXTENDED ENTRY EXAMPLE.....	C-1
INPUT LISTING.....	C-1
OUTPUT LISTING.....	C-3





# FOREWORD

## WHAT IS DECIBLE III

DECIBLE III is a preprocessor that is used to translate decision tables into optimized compileable coding. It also contains a source language library maintenance system and a short-hand translation system.

## LANGUAGE

There are three versions of DECIBLE III available; a COBOL, a PL/1, and a FORTRAN version. All versions are written in DECIBLE III produced COBOL coding. This manual is intended for use with the COBOL version. Separate manuals for the PL/1 and FORTRAN versions are available.

## HARDWARE REQUIREMENTS

DECIBLE III is available on any computer offering a standard COBOL compiler with a minimum available core size that is the equivalent of 64k bytes or 28k bytes plus overlay capability. For utilization of the source language library maintenance facilities, two tape drives or one disc drive is required.

## TRAINING

INDEPENDENCE COMPUTING & SOFTWARE CORPORATION provides all users, as part of the installation of DECIBLE III, a complete training course. This training course generally consists of complete training in the use of decision tables and the use of DECIBLE III.

## THIS MANUAL

This manual is intended to be a user training and reference guide. An understanding of the COBOL programming language and decision tables is assumed.





# 1. INTRODUCTION

## USE OF DECISION TABLES

Decision tables were developed as a tool for communicating logical procedures from person to person. Their superiority over narrative descriptions and flowcharts is rapidly becoming recognized. Narrative descriptions are usually difficult to follow, tend to be either ambiguous or incomplete, and are easily misinterpreted. Flowcharts are more exact than the narrative description but are more difficult to prepare, tend to be quite bulky and hard to follow, and, because certain tests or conditions must be shown more than once, can be error prone. In addition, flowcharts are usually very inflexible and difficult to change. A simple change in the logic of a problem may cause rewriting pages of flowcharts.

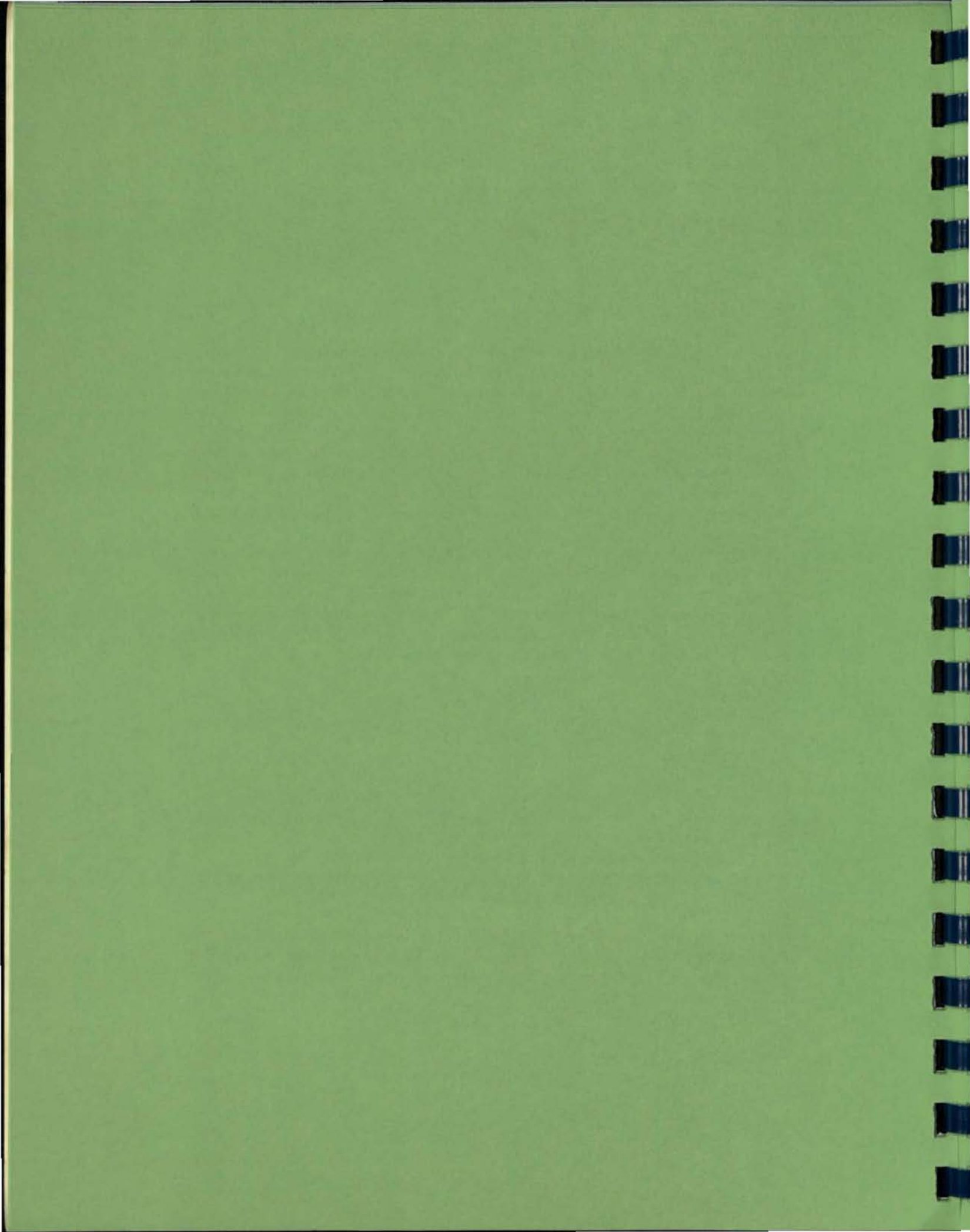
Decision tables, however, are not only easy to prepare, but can be read by anybody without special training. Most people, in fact, have worked with decision tables of one type or another; tax tables and mileage charts on road maps are good examples of a type of decision table. Logic that may require pages of flowcharts can generally be shown in one or two simple decision tables. By their very nature, decision tables preclude the most common logic errors — ambiguity and incompleteness. The format of decision tables enables even major changes to the logic to be made very easily.

## USE OF DECIBLE III

Decision tables can then be seen to be a major advance in the ability to communicate ideas and logic from person to person. DECIBLE III gives the user a means of taking this human communications tool and converting it directly into computer programs. By creating the programs directly from the definition of the problem, the difficulties of program logic errors, misinterpretations of problem definition, and lack of proper program documentation are solved.

Moreover, the coding produced by DECIBLE III is completely optimized, insuring that optimum coding for your installation will be produced from the decision tables. This optimization reduces the run time and core requirements of programs written in decision tables.







## 2. SYSTEM DESCRIPTION

### GENERAL DESCRIPTION

DECIBLE III accepts any combination of COBOL statements and decision tables and translates the decision tables into optimized compileable COBOL coding. The decision tables become NOTE paragraphs within the produced program and therefore provide complete documentation within a single listing.

Shorthand definitions are placed at the beginning of the program if they are to be used throughout the program (global definitions) or at the beginning of a decision table if they are to be used only within that table (local definitions).

### INPUT TO DECIBLE III

The input to DECIBLE III consists of the following:

- ◆ 80 column punched cards, or
- ◆ a magnetic tape source language file, or
- ◆ a disk source language file

If the input is a tape or disk source language file, cards can be used to update and change the program simultaneously with the processing of the program.

### OUTPUT OF DECIBLE III

The output of DECIBLE III consists of the following:

- ◆ printer listing of program and diagnostics, and
- ◆ compileable COBOL program on cards, magnetic tape, or disk, and
- ◆ (optionally) a source language file on tape or disk



## DECISION TABLE PROCESSING

DECIBLE III produces complete COBOL procedure division sections from each decision table. The section name used is the table name assigned by the user in the DECIBLE TABLE statement. Therefore, decision tables may be executed in the same manner as COBOL sections. The program can PERFORM a decision table, GO TO a decision table or pass into a decision table from the coding preceding the decision table.

## SHORTHAND PROCESSING

Shorthand abbreviations anywhere within the program are replaced by their definitions. If a definition does not fit in the record with the abbreviation, another record is created to continue the statement. The break occurs at the last space character that will fit on the record.

Shorthand abbreviations that appear within a decision table are translated in the produced coding but not within the body of the decision table.



### 3.

# SOURCE LANGUAGE LIBRARY MAINTENANCE SYSTEM

## PURPOSE OF SYSTEM

The DECIBLE III source language library maintenance system serves different purposes for different users. For those users whose current software does not include a source language file maintenance system, DECIBLE III provides a convenient means of keeping source language programs on tape or disk and greatly reduces the amount of card handling required to update and compile programs.

For those users whose current software includes a source language file maintenance system, the DECIBLE III library system provides maintenance facilities against the users files simultaneously with the processing of the decision tables. This eliminates the need of a separate maintenance and processor pass of the file and reduces the computer run requirements.

## GENERAL DESCRIPTION

The basic input to DECIBLE III can be specified as being on cards, tape, or disk. If the input is on tape or disk, cards can be used to update or change the program. A new program tape or disk file, including all changes, can be specified. The DECIBLE OPTION statement (see Chapter 6) is used to specify the input and output devices.

## SEQUENCE FIELD

The COBOL statement sequence field (columns 1 through 6) is used to control updates against a tape or disk file. All programs are automatically resequenced by a factor of 10. The listing produced shows the sequence number field of the input immediately to the left of the new sequence number. All cards input into the system must be in sequence, except for those cards used in a block add (see below). All out-of-sequence cards are indicated by three astericks (\*\*\*) to the left of the statement on the listing and are ignored.



## INSERTING RECORDS

When the input is specified as being on tape or disk, a card will be inserted into the file if it has a sequence number that is not the same as a sequence number on the tape. In the example below, two cards are to be added into the file between records 002420 and 002430:

### EXAMPLE

file input:

002420	A MOVE 10 TO COUNTER-A	4 4 2 2
002430	A RE-ENTER	6 6 4 4

card input:

002424	A MOVE COUNTER-A TO SUBSCRIPT-B	
002426		5 5 3 3

output listing (note resequencing):

002420	002630	A MOVE 10 TO COUNTER-A	4 4 2 2
002424	002640	A MOVE COUNTER-A TO SUBSCRIPT-B	
002426	002650		5 5 3 3
002430	002660	A RE-ENTER	6 6 4 4



## INSERTING A BLOCK OF RECORDS

Any card read with blanks (spaces) in the sequence field is assumed to follow the card immediately preceding it. In this way, a block of records can be inserted in a desired location. This is the only exception to the rule that all cards must be in sequence. In the following example, a block of records is to be inserted into a file.

### EXAMPLE

file input:

```
000840 03 FILLER PICTURE X(54).
000850 01 INPUT-RATE-TRANSACTION.
```

card input:

```
000845 01 MONEY-TRANSACTION-A.
03 CARD-TYPE PICTURE X.

*** (ADDITIONAL INPUT CARDS)

03 IDENTIFICATION-AREA PICTURE X(8).
```

output listing:

```
000840 000840 03 FILLER PICTURE X(54).
000845 000850 01 MONEY-TRANSACTION-A.
000860 03 CARD-TYPE PICTURE X.

***

000850 001020 03 IDENTIFICATION-AREA PICTURE X(8).
000850 001030 01 INPUT-RATE-TRANSACTION.
```



## REPLACING AND DELETING RECORDS

When the sequence number of a card is the same as a sequence number on the input file, the record on the input file is deleted. If columns 7 through 80 on the card contains all blanks (spaces), the card acts just as a delete card. If there is any non-blank in any of the columns 7 through 80 on the input card, that card replaces the file record. The following example shows a record being replaced and a record being deleted.

### EXAMPLE

file input:

```
001070 01  COUNTERS.
001080 03  NUMBER-RECORDS      PICTURE 9(5).
001090 03  NUMBER-FIELDS      PICTURE 9(5).
001100 03  NUMBER-ERRORS COMPUTATIONAL-3
001110                                PICTURE 9(5).
```

card input:

```
001080 03  NUMBER-RECORDS COMPUTATIONAL-3
001085                                PICTURE 9(5).
001090
```

output listing:

```
001070 001070 01  COUNTERS.
001080 001080 03  NUMBER-RECORDS COMPUTATIONAL-3
001085 001090                                PICTURE 9(5).
001100 001100 03  NUMBER-ERRORS COMPUTATIONAL-3
001110 001110                                PICTURE 9(5).
```



# 4. SHORT-HAND TRANSLATION SYSTEM

## GENERAL DESCRIPTION

The DECIBLE III shorthand system permits three character abbreviations for programmer selected phrases. Global abbreviations are defined at the beginning of the program and may be used anywhere within the program. Local abbreviations are defined in a decision table and may only be used for that table.

All abbreviations are three characters long preceded by a semi-colon (;) and may contain any alphanumeric or special character including spaces. All global abbreviations must be unique; local abbreviations can be used in different tables with different definitions.

The characters in the definition will replace the semi-colon and three character abbreviation. If the number of characters exceeds the amount permitted on a line, the line is continued on a new record. The continuation occurs at the last space that will fit on the line.

Abbreviations used in a decision table will be translated in the DECIBLE III produced coding but not in the decision table itself. This helps maintain the format and clarity of the decision table.

## ABBREVIATION DEFINITION STATEMENT

The ABBREVIATION DEFINITION statements are used to define DECIBLE III shorthand abbreviations. Global definition statements are the first records following the DECIBLE OPTION statement. Local definitions immediately follow the DECIBLE TABLE statement.

The definitions may be one to fifty-four characters long and may contain any characters in the COBOL character set. The definition is contained within quotes. If a quotation mark is desired as part of the definition, it is indicated by two quotation marks in the definition. One definition may not contain another. See Chapter 6 for a description of the ABBREVIATION DEFINITION statement.



## EXAMPLES OF ABBREVIATION DEFINITIONS

### EXAMPLE 1.

DEFINITION: ;NEF 'IS NOT EQUAL TO "FINISH"'  
STATEMENT: ITEM-PROCESS ;NEF  
GENERATES: ITEM-PROCESS IS NOT EQUAL TO 'FINISH'

### EXAMPLE 2.

DEFINITION: ;NUM 'COMP-3 VALUE 0 PICTURE S9(5)V99'  
DEFINITION: ;SAL 'EMPLOYEE-WEEKLY-SALARY'  
STATEMENT: 03 ;SAL ;NUM.  
GENERATES:  
03 EMPLOYEE -WEEKLY-SALARY COMP-3 VALUE 0 PICTURE S9(5)V99.





# 5. WRITING DECIBLE III DECISION TABLES

## GENERAL DESCRIPTION

Writing decision tables for DECIBLE III is similar to writing decision tables that are not processed automatically, except that all condition statements are valid COBOL conditional statements, and all action statements are valid COBOL procedure division statements.

## EXTENDED AND MIXED ENTRY DECISION TABLES

DECIBLE III permits the use of decision tables containing extended entry statements. These tables may consist entirely of extended entry statements (extended entry decision tables) or a combination of extended entry statements and limited entry statements (mixed entry decision tables).

In an extended entry statement the rule entry consists of the continuation of the action or condition statement. While the length of the rule entry (in mixed or extended entry tables) is fixed at four characters, the use of the Shorthand Translation System (see Chapter 4) adds greatly to the flexibility of extended entry tables.

## CODING INSTRUCTIONS

DECIBLE III decision tables can be coded on any standard COBOL coding sheets or on the coding sheets provided by ICS Corporation. (See page 16 for a sample coding sheet). On all decision table records, columns 7 through 11 and column 72 must contain a blank (space) character.

Extended or mixed entry decision tables can have a maximum of twenty records for individual action or condition statements. Limited entry decision tables have no maximum number of records for an individual statement. In any case, the total number of records (exclusive of COMMENT statements) in a single table may not exceed the limit set for your particular installation (usually one hundred records).





## DECISION TABLE IDENTIFICATION

Each decision table is assigned a name and a number (see the DECIBLE TABLE statement). The user may assign any valid COBOL procedure name up to thirty characters long (if the COBOL compiler permits). As this name becomes the COBOL section name, all tables must have a unique name. If no name is assigned by the user, DECIBLE III will assign one.

The decision table number is used to provide unique paragraph names within the produced coding and for user documentation. All tables within a program must have a unique table number. If the user does not assign a table number, DECIBLE III assigns them in descending order starting with 9999. Table numbers may be one to four digits.

## CONDITION STATEMENTS

The condition statements can be any valid COBOL conditional statement except that the word 'IF' is left out. They can be as complex as required and contain any combination of 'AND' or 'OR' qualifiers as the COBOL compiler allows. They must not, however, contain any imbedded nested IF statements.

## ACTION STATEMENTS

The action statements may be any valid COBOL sentences containing as many statements as required. They may be as complex as the COBOL compiler allows, but they must not contain conditional statements. There should be no periods (.) in the action statements except as non-numeric literals or as decimal points within numeric literals.

## INITIAL SET ACTIONS

Initial set actions are actions to be performed immediately upon entering the table and prior to the testing of the conditions. They can be used for setting counters and switches, reading input, etc. They may be any valid COBOL statements except conditional statements, GO TO statements or DECIBLE III special actions (see below).

An important aspect of initial set actions is that they are standard decision table actions with the added feature that they are also performed prior to testing the conditions. Therefore, they are coded with the other actions within the table and may have entries within the rules and be used the same as any other actions. The initial set actions are performed in the order coded.



## DECIBLE III SPECIAL ACTION STATEMENTS

In order to facilitate the use of DECIBLE III decision tables in looping operations and to provide a common ending point, DECIBLE III provides the following special action statements:

### 1. *LOOP*

This action produces coding to branch back to the beginning of the condition testing logic. If there are any initial set actions present they will not be executed again.

### 2. *RE-ENTER*

This action produces coding to branch back to the beginning of the decision table. Any initial set actions present will be executed again.

### NOTE:

As the distinction between the *LOOP* and *RE-ENTER* action statements concerns whether or not the initial set actions will be re-executed, the *LOOP* and *RE-ENTER* actions should not both be used in a decision table that does not contain an initial set rule.

### 3. *EXIT*

This action produces coding to branch to the end of the decision table. If the table was called by the COBOL 'PERFORM' verb, control passes to the statement following the PERFORM statement; otherwise control passes to the coding or decision tables following that decision table.

## ELSE RULE

The else rule is an optional rule whose actions are to be performed only if none of the other rules can be satisfied. Normally, rules must satisfy every possible combination of conditions, but in some cases the user of decision tables is interested only in certain specified sets of conditions. For example, a decision table used for validity checking may have hundreds of different combinations of conditions, only some of which are invalid. The standard rules may explicitly show the invalid combinations, with the ELSE rule handling all the valid combinations.

The ELSE rule is indicated by having the right most rule in the table have condition entries consisting of all dashes (-).



## END OF DECISION TABLE

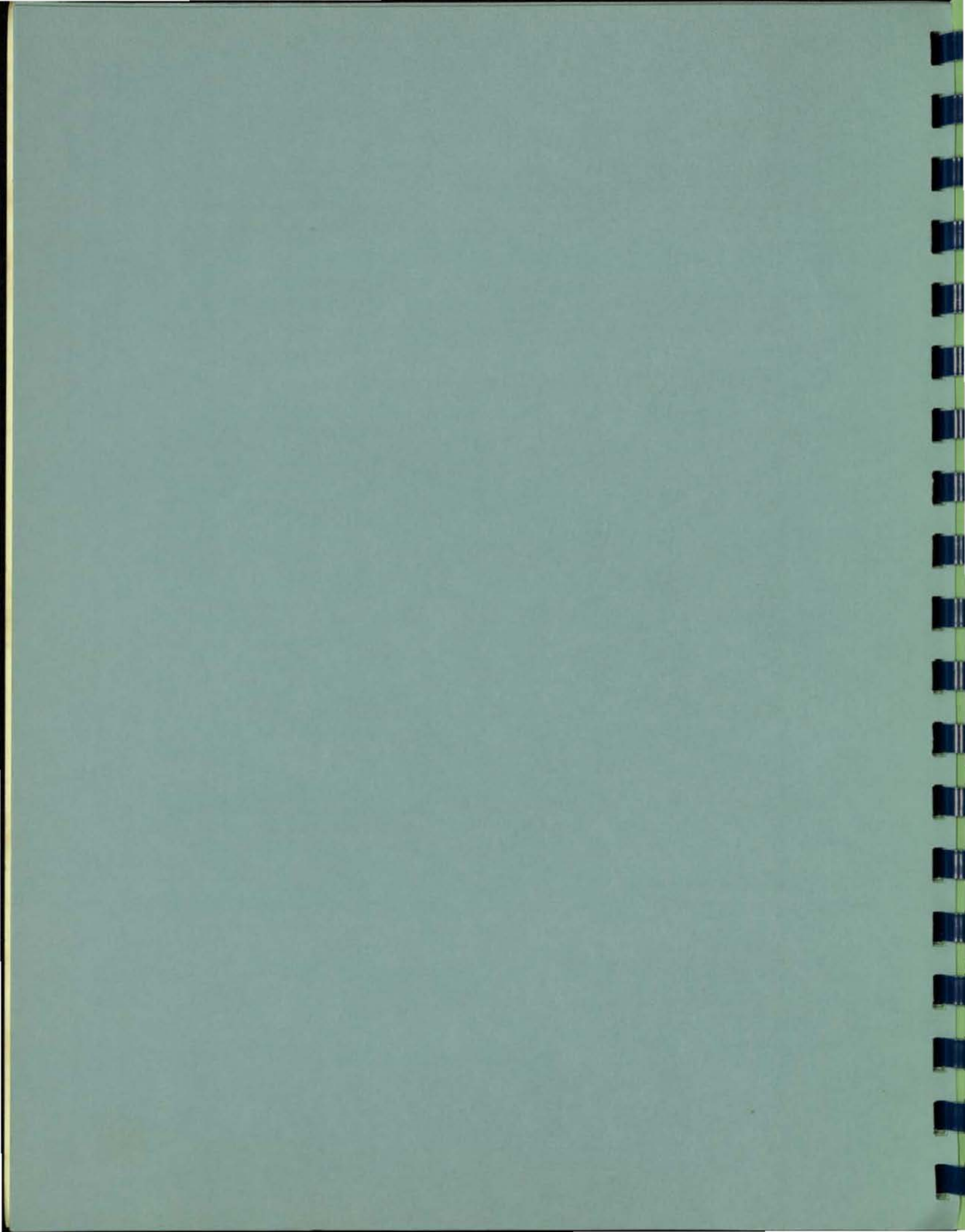
The end of the decision table is indicated by any statement that does not fit the format of a DECIBLE III statement. Specifically, this would be any card with a non-blank character in columns 8 through 11, or any character other than a C, A, S, \*, or space in column 12, or an end-of-file indicator.

Therefore, a procedure name (in margin A of the standard COBOL coding format), a COBOL statement starting in column 12, another decision table, or an end of file indicates the end of a decision table.

## LIMITED/EXTENDED ENTRY COMPARISON

ITEM	LIMITED ENTRY TABLES	EXTENDED OR MIXED ENTRY TABLES
DECIBLE TABLE STATEMENT		same as limited entry
X STATEMENT	Must not be used	Must be used
EXTENDED ENTRY STATEMENT	Must not be used	Must be used
WIDTH OF RULE	2 Columns	4 Columns
CONDITION RULE ENTRIES	Y, N, or dash (-)	4 characters, NONE, or dash for extended entry or Y, N, or dash
ACTION RULE ENTRIES	X, dash, or numeric value	4 characters or dash for extended entry or X or dash
NUMBER OF CARDS PER STATEMENT	Unlimited	20 card limit
RULES CONTINUE ON ADDITIONAL CARD	NO	YES
RULES START AT	Free format	Column 30





# 6. DECIBLE III STATEMENTS

## GENERAL DESCRIPTION

Three statement card types plus a comments card are used to code decision tables in DECIBLE III. They are the DECIBLE TABLE statement (used to identify decision tables), the X statement (used to identify a decision table as extended or mixed entry), and the DECIBLE SET statement (used to code condition and action statements).

Two statement card types are used by the source language library system and the shorthand translation system. They are the DECIBLE OPTION statement (used to set the input/output options) and the ABBREVIATION DEFINITION statement (used to define shorthand abbreviations).

## DECIBLE OPTION STATEMENT

The DECIBLE OPTION statement is used to set the input/output options. It can also be used to control the printer listing produced during the DECIBLE run. The options can be listed in any order desired.

## STATEMENT FORMAT

COLUMN	CONTENTS
1 - 6	sequence number
12 - 23	NOTE OPTION
24 - 49	options, separated by commas
50 - 72	listing heading

The listing heading field in the DECIBLE OPTION statement is printed at the top of each page in the listing. Any subsequent option statement with any entry other than spaces in the listing heading field will cause the listing to skip to the top of a page and the new heading will replace the one previously entered.



If the listing heading field in a DECIBLE OPTION statement consists of all spaces, the previous heading entered remains and no skip to the top of the page occurs.

If the first card in the card reader is not an option card, the input is assumed to be on cards and the run is assumed to be a syntax check run only, producing only a printer listing output. The following options are used only when the DECIBLE OPTION card is the first card in the card reader:

*TAPE*

This option indicates that the primary input comes from a source language file on tape.

*DISK*

This option indicates that the primary input comes from a source language file on disk. The TAPE and DISK options cannot both be used in the same program. In the absence of both of the two options, the primary input file is assumed to be on cards.

*NEWT*

This option indicates that a new source language file is to be created on tape.

*NEWD*

This option indicates that a new source language file is to be created on disk. The NEWT and NEWD options cannot both be used in the same program.

*COMC*

This option indicates that the compileable program produced by DECIBLE III is to be punched on cards.

*COMT*

This option indicates that the compileable program is to be created on tape in a format acceptable to the COBOL compiler for the system in which DECIBLE III is implemented.

*COMD*

This option indicates that the compileable program is to be created on disk in a format acceptable to the COBOL compiler for the system in which DECIBLE III is implemented. It should be noted that some operating systems do not permit the input to the COBOL compiler to come from disk. When DECIBLE III is implemented on these systems, the COMD option will not be accepted.





Only one of the options COMC, COMT, or COMD can be used in a DECIBLE III run. In the absence of any of these options, DECIBLE III operates in syntax check mode only.

The following options can be used anywhere within the program. They are used to control the printer listing.

*NOPR*

This option stops the DECIBLE III printer listing. All diagnostic messages, along with the input line previous to the diagnostic, will still be printed.

*PRNT*

This option causes DECIBLE III to resume the listing if the NOPR option had been previously entered.

#### EXAMPLES OF DECIBLE OPTION STATEMENT

1. Input on cards, no library, compileable output on tape  
000010 NOTE OPTION COMT
2. Input on cards, create tape library file, syntax check only  
000010 NOTE OPTION NEWT
3. Input on tape, compiler output on tape, no listing  
000010 NOTE OPTION NOPR, COMT, TAPE
4. Input on disk, create new disk file, compileable output on tape  
000010 NOTE OPTION DISK, NEWD, COMT
5. Input on disk, create tape library file, compileable output on disk  
000010 NOTE OPTION NEWT, DISK, COMD



## DECIBLE TABLE STATEMENT

This statement is used to indicate the beginning of a decision table. The table name may be any valid COBOL PROCEDURE name up to 30 characters long. If a table number is assigned by the user, the name is immediately followed by a comma and the table number. The table number may be one to four digits long. All tables within a program must have unique table numbers.

If no table number is assigned, then DECIBLE III assigns them in descending order, starting with 9999.

### STATEMENT FORMAT

COLUMN	CONTENTS
1 - 6	sequence number
12 - 23	NOTE DECIBLE
25 - 72	in free format, table name followed (if desired) by a comma and a table number



### EXAMPLES OF DECIBLE TABLE STATEMENT

1. DECIBLE III assigns unique table name and number  
009320 NOTE DECIBLE
2. User assigns table name , DECIBLE III assigns table number  
009320 NOTE DECIBLE CALCULATE-PAY-RATE
3. User assigns table number, DECIBLE III assigns table name  
009320 NOTE DECIBLE, 136
4. User assigns table name and number  
009320 NOTE DECIBLE CALCULATE-PAY-RATE, 136



## X STATEMENT

The X statement is used only in extended or mixed entry tables to indicate that the table is not a limited entry decision table and to give the number of rules within the table. It immediately follows the DECIBLE TABLE statement.

The X statement contains an X in column 12 of the card and the number of rules (right justified) in columns 14 and 15. Columns 17 through 72 may contain comments if desired.

### STATEMENT FORMAT

COLUMN	CONTENTS
1 - 6	sequence number
12	X
14 - 15	number of rules in table
17 - 72	comments



## DECIBLE SET STATEMENT

The DECIBLE SET statement is used to code condition and action statements. Limited entry condition and action statements can be continued on as many records as required; extended entry statements have a limit of 20 records. The maximum total number of statement records in a table is 100, not counting comment cards.

### STATEMENT FORMAT

COLUMN	CONTENTS
1 - 6	sequence number
12	set type (on first line only)
14 - 71	statement, followed by rule entries
72	space

The set type (column 12) indicates whether the set is a condition (C), action (A), or initial set action (S). The extended entry indicator (column 13) indicates whether the set is limited (blank) or extended (X). Extended entry sets can only be used in an extended or mixed entry table.

Rules in a limited entry table may start in any column from 30 through 70 of the last card of a set. Limited entry tables use two columns per rule. Valid condition rule entries are Y, N, or dash (-) and valid action rule entries are X, dash (-), or any number from 1 to 99. All rule entries are right justified. Limited entry rule entries must be contained on the last record of a set.

Rules in an extended or mixed entry table are four columns wide and must start in column 30. If more than ten rules are used, the remaining rules are coded in an additional record, again starting in column 30. Limited entry condition entries coded in a mixed entry table must be Y, N, or dash (-) and limited entry action entries must be X or dash (-). Note that action sequence numbers are not permitted in a mixed entry table.



In extended entry statements, the rule entry is a dash (-), the continuation of the condition or action, or DECIBLE special entries (see Chapter 5). The entries may be one to four characters long and right justified within the rule. DECIBLE shorthand abbreviations are valid rule entries.

Conditions or actions must end at least two columns before the beginning of the rule entries. On all records of a set, column 72 must be blank. Special care should be used in the coding of the first set as this is the one DECIBLE III uses to determine the position and number of rules in limited entry tables. Rule entries should never be all spaces.

#### DECIBLE COMMENT STATEMENT

Comment cards may be placed anywhere within a decision table.

#### STATEMENT FORMAT

COLUMN	CONTENTS
1 - 6	sequence number
12	asterisk (*)
14 - 72	any comments



## ABBREVIATION DEFINITION STATEMENT

The ABBREVIATION DEFINITION statements are used to define DECIBLE III shorthand abbreviations. They must immediately follow the first DECIBLE OPTION statement.

The definitions may be 1 to 54 characters long and contain any characters in the COBOL character set. The definition is contained within quotation marks. Any quotation mark embedded within the definition is indicated by two quotation marks. One definition may not contain another (nesting).

Abbreviation definitions are copied on the DECIBLE III library system file, but do not appear in the COBOL compilation listing.

### STATEMENT FORMAT

COLUMN	CONTENTS
1 - 6	sequence number
12	semi-colon (;)
13 - 15	abbreviation
17 - 72	in free format, the definition contained within quotes

### EXAMPLES OF ABBREVIATION DEFINITIONS

#### EXAMPLE 1.

```
DEFINITION: ;NEF 'IS NOT EQUAL TO "FINISH"'
```

```
STATEMENT: ITEM-PROCESS ;NEF
```

```
GENERATES: ITEM-PROCESS IS NOT EQUAL TO 'FINISH'
```

#### EXAMPLE 2.

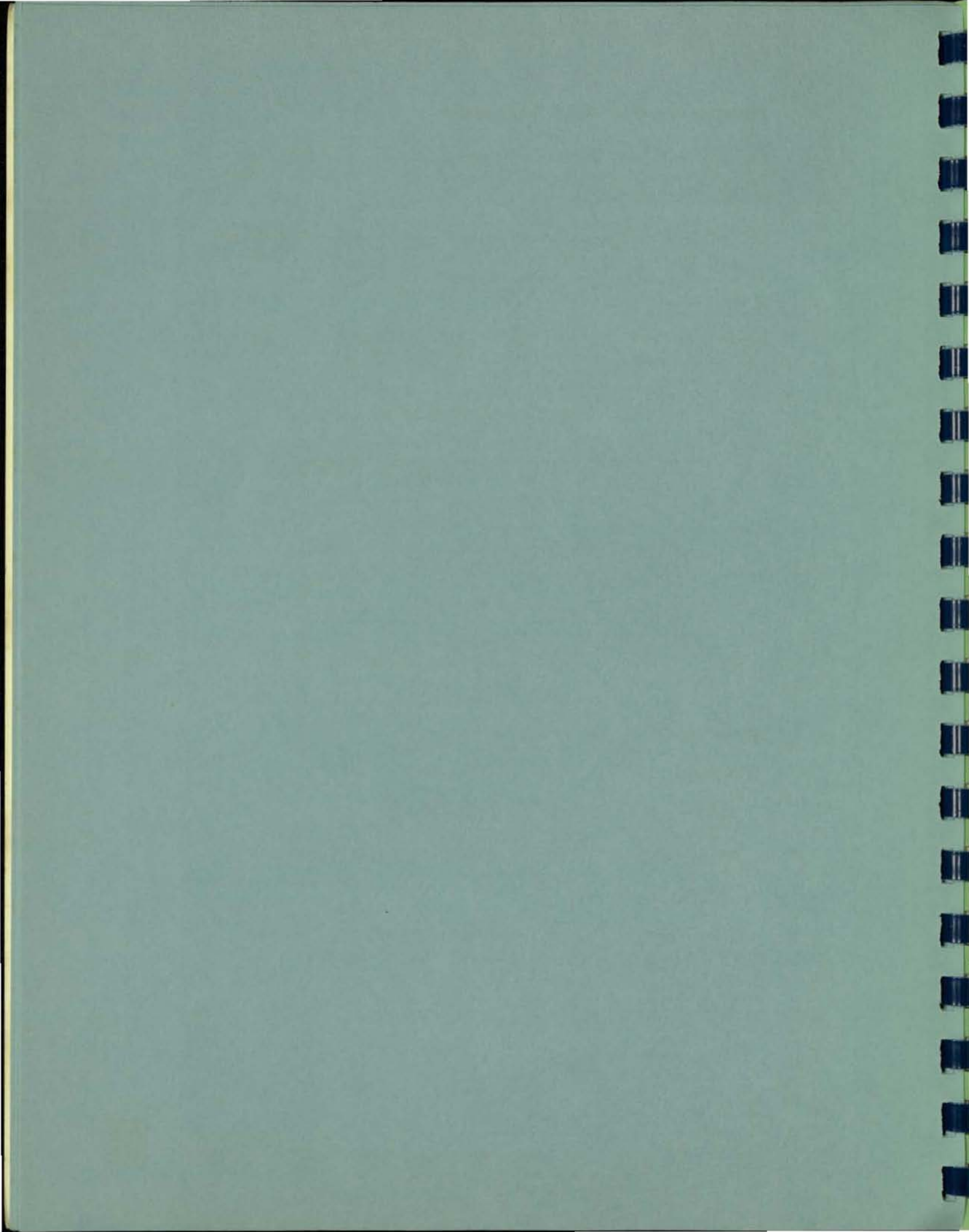
```
DEFINITION: ;NUM 'COMP-3 VALUE 0 PICTURE S9(5)V99'
```

```
DEFINITION: ;SAL 'EMPLOYEE-WEEKLY-SALARY'
```

```
STATEMENT: 03 ;SAL ;NUM.
```

```
GENERATES: 03 EMPLOYEE-WEEKLY-SALARY COMP-3 VALUE 0 PICTURE S9(5)V99.
```







# 7. PROGRAMMING GUIDELINES

## TABLE ORDER

The algorithm used to translate the decision table is completely independent of the order of the rules except that the "ELSE" rule must always be the last rule in the table. The produced coding will test the rules in the most efficient order to reduce running time and core requirements. Only if it is immaterial to the efficiency of the produced coding will DECIBLE III test the conditions in the order coded. It should be noted that in different branches of the condition testing logic, the same conditions may be tested in a different order.

The actions are executed in the order of their sequence number and, except for the initial set actions, it is immaterial in which order they are coded. The initial set rule is executed in the order coded.

In order to take full advantage of the action optimization routines, if a set of actions is performed by more than one rule, these rules should perform those actions in the same order.

## TABLE UNIQUENESS

In order to assure the unique generation of procedure names, no two tables should have the same name or number. Also no coding produced by the user should have a procedure or data name the same as a table name or begin with 'Dnnnn', where nnnn is a four digit number.

## VALUE RANGES

### *NUMBER OF TABLES*

DECIBLE III, will handle 9999 tables; each table must have a unique table number and table name.



#### *NUMBER OF RULES*

The minimum number of rules is 1.

The maximum number of rules (including the ELSE Rule if specified) is 20.

#### *NUMBER OF CONDITIONS*

The minimum number of condition statements is zero.

The maximum number of condition statements is 20.

#### *NUMBER OF ACTIONS*

The minimum number of action statements is 2.

The maximum number of action statements is 30.

#### *NUMBER OF CARDS*

A decision table may contain up to 100 cards, not including the DECIBLE TABLE card and COMMENT cards.

### CONDITION STATEMENTS

Condition statements may be any valid COBOL conditional statement except that they may not contain nested IF statements. They may be complex conditions and contain any combination of "AND" and "OR" statements permitted by the compiler.

### ACTION STATEMENTS

Action statements may be any valid COBOL statements except conditional statements.

### RULES

All rules must contain at least one action statement and the last action to be executed within each rule (the entry with the highest action sequence number) except the initial set rule must be a "GO TO", "RE-ENTER", "LOOP", "EXIT", or "STOP RUN".



# APPENDIX A

## DIAGNOSTICS

There are two different types of diagnostics – error and warning messages. In addition, all input cards out of sequence are indicated by printing three astericks (\*\*\*) to the left of the input card. All cards out of sequence are ignored.

A warning message indicates that an error condition exists that the system can correct. In some cases these corrections may involve assumptions about the intent of the programmer and may be incorrect. All warning messages should be checked to see that corrections made truly reflect the intentions of the programmer.

An error message indicates that an unrecoverable error has been made and further processing of a table is impossible. In that case, the section produced by DECIBLE III will contain only the COBOL note paragraph containing the decision table itself.

All error and warning messages contain a number key that can be used to reference this section for further information. The following is a list of all warning and error messages:

### *01 MORE THAN ONE LIBRARY INPUT FILE OPTION*

The first DECIBLE OPTION statement contains both the TAPE and DISK options.

### *02 MORE THAN ONE LIBRARY OUTPUT FILE OPTION*

The first DECIBLE OPTION statement contains both the NEWT and NEWD options.

### *03 INCORRECT NUMBER OF ACTIONS*

This message indicates that the table contains less than 2 or more than 30 actions. It may be caused by a card with other than a C, A, S, \*, or blanks in column 12 within the table or coding in margin A. In either case, DECIBLE III takes this as the end of the table.

### *04 INCORRECT NUMBER OF CONDITIONS*

The table contains more than 20 conditions.



- 05 *NO RULES IN FIRST STATEMENT*  
The first DECIBLE SET STATEMENT contains no rule entries.
- 06 *INVALID ACT STATEMENT or  
INVALID COND STATEMENT*  
The DECIBLE SET STATEMENT did not contain a valid action or condition statement.
- 07 *INVALID ENTRY ACT RULE nn or  
INVALID ENTRY COND RULE nn*  
The rule entry for an action was not a numeric value, X, or dash (-) or the rule entry for a condition was not Y, N, or dash. 'nn' is the rule number with the invalid entry. DECIBLE III assumes a dash (-) entry for this rule.
- 08 *INCOMPLETE TABLE ELSE RULE ASSIGNED*  
There is no ELSE rule, but all possible conditions have not been accounted for. An ELSE rule containing a 'STOP RUN' action is created.
- 09 *RULES nn AND mm NOT UNIQUE*  
The same set of conditions will pass rules 'nn' and 'mm'.
- 10 *'IF' STATEMENT NOT ACCEPTED AS ACTION*  
A conditional statement is not a valid action.
- 11 *LOOP AND RE-ENTER USED - NO INITIAL SET*  
Since the only distinction between the special actions LOOP and RE-ENTER involve the initial set, they should not both be used in a table with no initial set.
- 12 *NO ACTIONS IN RULE nn*  
Rule 'nn' contains no actions. A 'STOP RUN' action is added to that rule.
- 13 *COLUMN nn CONTAINS INVALID CHARACTER*
- 14 *MISSING TABLE NAME*  
DECIBLE assigns a table number.



**15 MORE THAN 10 RULES NEED CONTINUATION**

In an extended entry table, there is only room for ten rules per card. Therefore, more than ten rules require the rules to be contained on more than one card. The action or condition flagged by this message did not have the required number of cards.

**16 aa ERRORS bb WARNINGS cc TABLES**

This message appears five times at the end of every program and gives the total number of errors, warnings, and tables in the program.

**17 SYSTEM DIAGNOSTIC**

Notify Independence Computing and Software Corp. immediately should this message occur.

**18 NUMBER OF RULES LOGICALLY INCORRECT**

A decision table with no conditions contains more than one rule, or a decision table with one condition contains other than two rules.

**19 CONDITION STATEMENT OUT OF ORDER**

A condition statement follows an action statement in a table.

**20 MORE THAN 100 CARDS IN TABLE**

**21 aaaa IS AN UNKNOWN OPTION**

The DECIBLE OPTION STATEMENT contains the unknown option 'aaaa'.

**22 MORE THAN 20 CARDS IN EXTENDED STATEMENT**

DECIBLE SET STATEMENTS used in extended entry tables may contain no more than 20 cards.

**23 DUPLICATE SHORTHAND ABBREVIATION**

This abbreviation has been previously defined.

**24 BEGINNING QUOTE OF DEFINITION MISSING**

Definition must be enclosed in quotes.



25 *END QUOTE OF DEFINITION MISSING*  
Definition must be enclosed in quotes.

26 *UNDEFINED SHORTHAND ABBREVIATION ;aaa*  
The abbreviation ;aaa used in a statement has not been defined  
in an ABBREVIATION DEFINITION STATEMENT.

27 *SHORTHAND STACK OVERFLOW*  
There are more than 600 shorthand definitions or their combined  
length is greater than 15,000 characters.

28 *ACTION ENTRY IN RULE nn GREATER THAN 90*  
Sequenced action entries can not be greater than 90.



```

000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID. UPDATE.
000030 AUTHOR. CHARLES STERBAKOV.
000040 ENVIRONMENT DIVISION.
000050 CONFIGURATION SECTION.
000060 SOURCE-COMPUTER. IBM-360.
000070 OBJECT-COMPUTER. IBM-360.
000080 INPUT-OUTPUT SECTION.
000090     SELECT CARDIN
000100     ASSIGN TO UT-S-CARDIN.
000110     SELECT TAPEIN
000120     ASSIGN TO UT-S-TAPEIN.
000130     SELECT TAPEOUT
000140     ASSIGN TO UT-S-TAPEOUT.
000150 DATA DIVISION.
000160 FILE SECTION.
000170 FD CARDIN
000180     RECORDING MODE F
000190     LABEL RECORDS OMITTED
000200     DATA RECORD IS CARD-RECORD.
000210     01 CARD-RECORD.
000220     03 CARD-SEQUENCE             PICTURE 9(6).
000230     03 CARD-DATA-AREA             PICTURE X(66).
000240     03 FILLER                     PICTURE X(8).
000250 FD TAPEIN
000260     RECORDING MODE F
000270     BLOCK 5 RECORDS
000280     LABEL RECORDS STANDARD
000290     DATA RECORD IS INPUT-RECORD.
000300     01 INPUT-RECORD.
000310     03 TAPE-SEQUENCE             PICTURE 9(6).
000320     03 FILLER                     PICTURE X(74).
000330 FD TAPEOUT
000340     RECORDING MODE F
000350     BLOCK 5 RECORDS
000360     LABEL RECORDS STANDARD
000370     DATA RECORD IS OUTPUT-RECORD.
000380     01 OUTPUT-RECORD.
000390     03 OUTPUT-SEQUENCE           PICTURE 9(6).
000400     03 FILLER                     PICTURE X(74).
000410 WORKING-STORAGE SECTION.
000420 MISC-DATA.
000430     03 LAST-OUT-SEQUENCE     VALUE 10     COMPUTATIONAL-3
000440     03 CARD-FILE-EOF VALUE 'N'     PICTURE 9(6).
000450     03 TAPE-FILE-EOF VALUE 'N'     PICTURE X.
000460     03 TAPE-FILE-EOF VALUE 'N'     PICTURE X.
000470 PROCEDURE DIVISION.
000480 INITIALIZE SECTION.
000490 INITIAL.
000500     OPEN INPUT CARDIN
000510     INPUT TAPEIN
000520     OUTPUT TAPEOUT.
000530     NOTE DECIBLE MERGE-ROUTINE,5
000540     C CARD-FILE-EOF = 'Y'           Y Y'N N N N N
000550     C TAPE-FILE-EOF = 'Y'           Y N Y N N N N
000560     C CARD-SEQUENCE IS LESS THAN TAPE-SEQUENCE
    
```

B-1

APPENDIX B  
INPUT LISTING

```

000570          - - - Y N N N
000580 C CARD-SEQUENCE IS EQUAL TO TAPE-SEQUENCE
000590          - - - Y Y N
000600 C CARD-DATA-AREA = SPACES - - - Y N -
000610 *
000620 S READ CARDIN
000630   AT END MOVE 'Y' TO
000640   CARD-FILE-EDF      - - 4 4 - - -
000650 S READ TAPEIN
000660   AT END MOVE 'Y' TO
000670   TAPE-FILE-EDF      - 4 - - - - 4
000680 A CLOSE CARDIN TAPEIN TAPEOUT
000690   STOP RUN          2 - - - - -
000700 A MOVE INPUT-RECORD TO OUTPUT-RECORD
000710          - 2 - - - - 2
000720 A MOVE CARD-RECORD TO OUTPUT-RECORD
000730          - - 2 2 - 2 -
000740 A PERFORM WRITE-TAPE      - 6 6 6 - 4 6
000750 A RE-ENTER              - - - - 2 6 -
000760 A LOOP                  - 8 8 8 - - 8
000770 WRITE-TAPE SECTION.
000780 WRITE-TAPE-RECORD.
000790 MOVE LAST-OUT-SEQUENCE TO CUTPUT-SEQUENCE.
000800 WRITE OUTPUT-RECORD.
000810 ADD 10 TO LAST-OUT-SEQUENCE.

```

B-2



ICS DECIBLE III VER 3 GEN 4

000010	000010	IDENTIFICATION DIVISION.		
000020	000020	PROGRAM-ID. UPDATE.		
000030	000030	AUTHOR. CHARLES STERBAKOV.		
000040	000040	ENVIRONMENT DIVISION.		
000050	000050	CONFIGURATION SECTION.		
000060	000060	SOURCE-COMPUTER. IBM-360.		
000070	000070	OBJECT-COMPUTER. IBM-360.		
000080	000080	INPUT-OUTPUT SECTION.		
000090	000090	SELECT CARDIN		
000100	000100	ASSIGN TO UT-S-CARDIN.		
000110	000110	SELECT TAPEIN		
000120	000120	ASSIGN TO UT-S-TAPEIN.		
000130	000130	SELECT TAPEOUT		
000140	000140	ASSIGN TO UT-S-TAPEOUT.		
000150	000150	DATA DIVISION.		
000160	000160	FILE SECTION.		
000170	000170	FD CARDIN		
000180	000180	RECORDING MODE F		
000190	000190	LABEL RECORDS OMITTED		
000200	000200	DATA RECORD IS CARD-RECORD.		
000210	000210	01 CARD-RECORD.		
000220	000220	03 CARD-SEQUENCE	PICTURE 9(6).	
000230	000230	03 CARD-DATA-AREA	PICTURE X(66).	
000240	000240	03 FILLER	PICTURE X(8).	
000250	000250	FD TAPEIN		
000260	000260	RECORDING MODE F		
000270	000270	BLOCK 5 RECORDS		
000280	000280	LABEL RECORDS STANDARD		
000290	000290	DATA RECORD IS INPUT-RECORD.		
000300	000300	01 INPUT-RECORD.		
000310	000310	03 TAPE-SEQUENCE	PICTURE 9(6).	
000320	000320	03 FILLER	PICTURE X(74).	
000330	000330	FD TAPEOUT		
000340	000340	RECORDING MODE F		
000350	000350	BLOCK 5 RECORDS		
000360	000360	LABEL RECORDS STANDARD		
000370	000370	DATA RECORD IS OUTPUT-RECORD.		
000380	000380	01 OUTPUT-RECORD.		
000390	000390	03 OUTPUT-SEQUENCE	PICTURE 9(6).	
000400	000400	03 FILLER	PICTURE X(74).	
000410	000410	WORKING-STORAGE SECTION.		
000420	000420	MISC-DATA.		
000430	000430	03 LAST-OUT-SEQUENCE	VALUE 10 COMPUTATIONAL-3	
000440	000440		PICTURE 9(6).	
000450	000450	03 CARD-FILE-EOF VALUE 'N'	PICTURE X.	
000460	000460	03 TAPE-FILE-EOF VALUE 'N'	PICTURE X.	
000470	000470	PROCEDURE DIVISION.		
000480	000480	INITIALIZE SECTION.		
000490	000490	INITIAL.		
000500	000500	OPEN INPUT CARDIN		
000510	000510	INPUT TAPEIN		
000520	000520	OUTPUT TAPEOUT.		
	000530	MERGE-ROUTINE SECTION.		
	000540	D0005N.		
000530	000550	NOTE DECIBLE MERGE-ROUTINE,5		

OUTPUT LISTING

```

000540
000550
000560
000570
000580
000590
000600
000610
000620
000630
000640
000650
000660
000670
000680
000690
000700
000710
000720
000730
000740
000750
000760
000770
000780
000790
000800
000810
000820
000830
000840
000850
000860
000870
000880
000890
000900
000910
000920
000930
000940
000950
000960
000970
000980
000990
001000
001010
001020
001030
001040
001050
001060
001070
001080
001090
001100

C CARD-FILE-EOF = 'Y'
C TAPE-FILE-EOF = 'Y'
C CARD-SEQUENCE IS LESS THAN TAPE-SEQUENCE
C CARD-SEQUENCE IS EQUAL TO TAPE-SEQUENCE
C CARD-DATA-AREA = SPACES
*
S READ CARDIN
  AT END MOVE 'Y' TC
  CARD-FILE-EOF
S READ TAPEIN
  AT END MOVE 'Y' TO
  TAPE-FILE-EOF
A CLOSE CARDIN TAPEIN TAPEOUT
  STOP RUN
A MOVE INPUT-RECORD TO OUTPUT-RECORD
  2
A MOVE CARD-RECORD TO OUTPUT-RECORD
  2
A PERFORM WRITE-TAPE
A RE-ENTER
A LOOP
  END.
000055.
READ CARDIN
  AT END MOVE 'Y' TO
  CARD-FILE-EOF.
READ TAPEIN
  AT END MOVE 'Y' TO
  TAPE-FILE-EOF.
000005.
IF CARD-FILE-EOF = 'Y'
IF TAPE-FILE-EOF = 'Y'
  CLOSE CARDIN TAPEIN TAPEOUT
  STOP RUN
ELSE
  GO TO D000552,
000930
IF TAPE-FILE-EOF = 'Y'
  GO TO D000553.
IF CARD-SEQUENCE IS LESS THAN TAPE-SEQUENCE
  GO TO D000553.
IF CARD-SEQUENCE IS EQUAL TO TAPE-SEQUENCE
IF CARD-DATA-AREA = SPACES
  GO TO D00055
ELSE
  GO TO D000556.
001030 D000552.
001040 MOVE INPUT-RECORD TO OUTPUT-RECORD.
001050 READ TAPEIN
  AT END MOVE 'Y' TC
  TAPE-FILE-EOF.
001080 D000558.
001090 PERFORM WRITE-TAPE.
001100 GO TO D00005.

Y Y N N N N N
Y Y N N N N N
- - - Y N N N
- - - Y N N N
- - - Y N
- - - Y N
- - 4 4 - -
- 4 - - - 4
- 4 - - - 4
2 - - - -
- 2 - - - 2
- 6 6 - 4 6
- - - 2 6 -
- 8 8 8 - - 8

```

ICS DECIBLE III VER 3 GEN 4

001110 D000553.  
001120 MOVE CARD-RECORD TO OUTPUT-RECORD.  
001130 READ CARDIN  
001140 AT END MOVE 'Y' TO  
001150 CARD-FILE-EOF.  
001160 GO TO D000558.  
001170 D000556.  
001180 MOVE CARD-RECORD TO OUTPUT-RECORD.  
001190 PERFORM WRITE-RECORD.  
001200 GO TO D00055.  
000770 001210 WRITE-TAPE SECTION.  
000780 001220 WRITE-TAPE-RECORD.  
000790 001230 MOVE LAST-OUT-SEQUENCE TO OUTPUT-SEQUENCE.  
000800 001240 WRITE OUTPUT-RECORD.  
000810 001250 ADD 10 TO LAST-OUT-SEQUENCE.

\*\*\*\*\* 16 WARNING 0 ERRORS 0 WARNINGS 1 TABLES \*\*\*\*\*

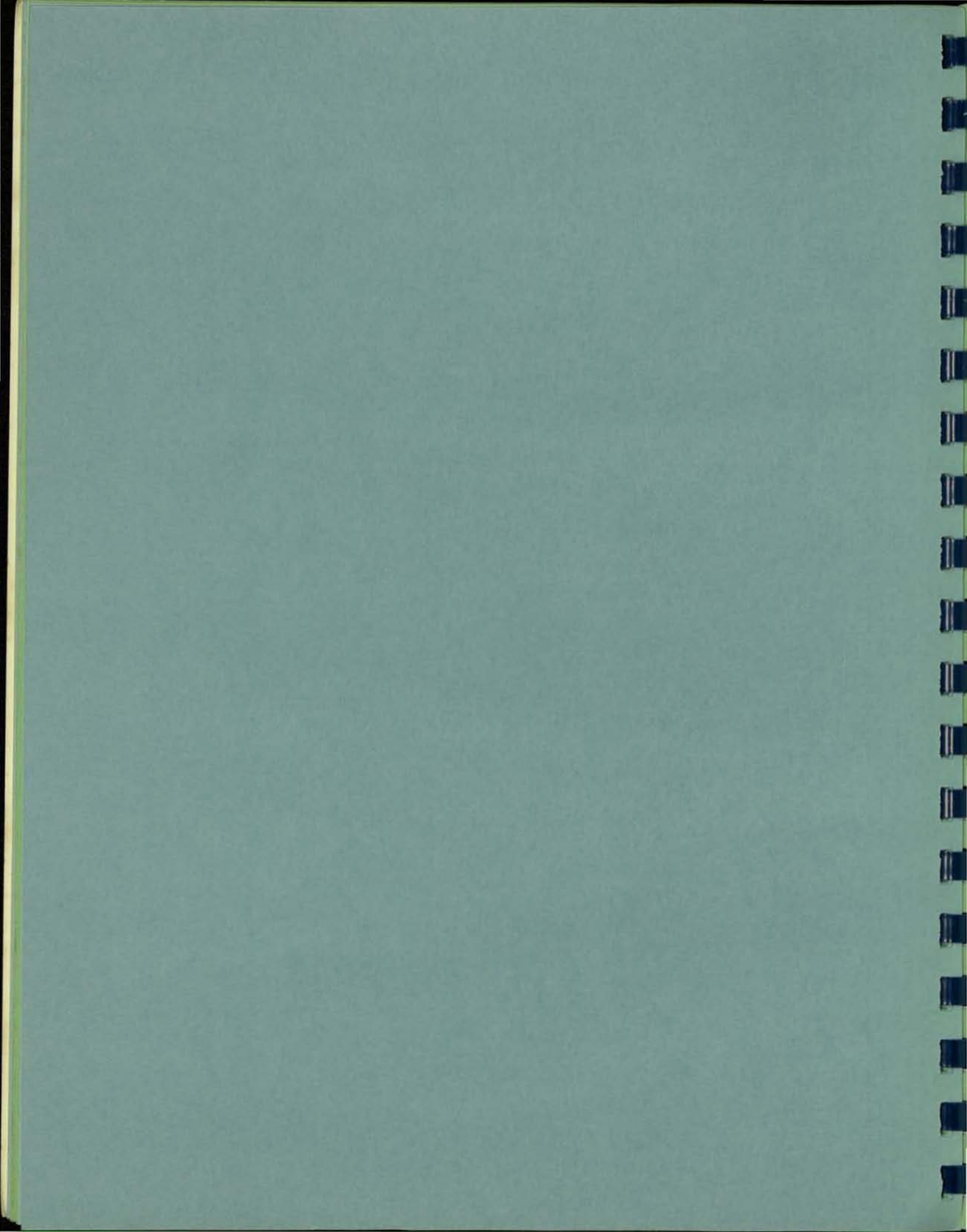
\*\*\*\*\* 16 WARNING 0 ERRORS 0 WARNINGS 1 TABLES \*\*\*\*\*

\*\*\*\*\* 16 WARNING 0 ERRORS 0 WARNINGS 1 TABLES \*\*\*\*\*

\*\*\*\*\* 16 WARNING 0 ERRORS 0 WARNINGS 1 TABLES \*\*\*\*\*

\*\*\*\*\* 16 WARNING 0 ERRORS 0 WARNINGS 1 TABLES \*\*\*\*\*

B-5



```

000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID. SORTKEY.
000030 AUTHOR. CHARLES STERBAKOV.
000040 ENVIRONMENT DIVISION.
000050 CONFIGURATION SECTION.
000060 SOURCE-COMPUTER. IBM-360.
000070 OBJECT-COMPUTER. IBM-360.
000080 INPUT-OUTPUT SECTION.
000090 FILE-CONTROL.
000100     SELECT INPUT-FILE
000110     ASSIGN TO UT-S-TAPEIN.
000120     SELECT OUTPUT-FILE
000130     ASSIGN TO UT-S-TAPOUT.
000140 DATA DIVISION.
000150 FILE SECTION.
000160 FD INPUT-FILE
000170     RECORDING MODE F
000180     BLOCK 10 RECORDS
000190     LABEL RECORDS STANDARD
000200     DATA RECORD IS INPUT-RECORD.
000210 01 INPUT-RECORD.
000220 03 FILLER                                PICTURE X(4).
000230 03 POGL-TYPE                             PICTURE X.
000240 03 FILLER                                PICTURE X(15).
000250 03 RISK-CODE                             PICTURE X.
000260 03 FILLER                                PICTURE X(19).
000270 03 ITEM-CODE                             PICTURE 99.
000280 03 FILLER                                PICTURE X(38).
000290 FD OUTPUT-FILE
000300     RECORDING MODE F
000310     BLOCK 10 RECORDS
000320     LABEL RECORDS STANDARD
000330     DATA RECORD IS INPUT-RECORD.
000340 01 DATA RECORD.
000350 03 EXPANDED-RECORD                       PICTURE X(80).
000360 03 SORT-KEY                             PICTURE 9.
000370 PROCEDURE DIVISION.
000380 INITIALIZE SECTION.
000390 START-PROGRAM.
000400     OPEN INPUT-FILE
000410     OUTPUT-FILE.
000420     NOTE DECIBLE SET-SORT-KEYS
000430     X 07
000440     CXPOGL-TYPE =      'N' 'N' 'S' 'T' 'T' 'T' -
000450     CXRISK-CODE =      'A' 'B' - 'A' 'C' 'C' -
000460     C ITEM-CODE = 15 OR 20 OR 30
000470     - - - - Y N -
000480     S READ INPUT-FILE
000490     AT END GO TO END-JOB
000500     - - - - -
000510     A MOVE 'A' TO RISK-CODE
000520     - - X - - -
000530     AXCCMPUTE SORT-KEY =
000540     1 2 1 3 4 5 0
000550     A MOVE INPUT-RECORD TO EXPANDED-RECORD
000560     WRITE OUTPUT-RECORD
    
```

C-1

APPENDIX C  
INPUT LISTING

```
000570  
000580 A RE-ENTER  
000590 END-JOB SECTION.  
000600 FINISH-JOB.  
000610 CLDSE INPUT-FILE WITH LOCK.  
000620 CLDSE OUTPUT-FILE.  
000630 STOP_RUN.  
  
X X X X X X X X  
X X X X X X X X
```

# OUTPUT LISTING

```

000010 000010 IDENTIFICATION DIVISION,
000020 000020 PROGRAM-ID, SORTKEY,
000030 000030 AUTHOR, CHARLES STERBAKOV,
000040 000040 ENVIRONMENT DIVISION,
000050 000050 CONFIGURATION SECTION,
000060 000060 SOURCE-COMPUTER, IBM-360,
000070 000070 DEJECT-COMPUTER, IBM-360,
000080 000080 INPUT-OUTPUT SECTION,
000090 000090 FILE-CONTROL,
000100 000100 SELECT INPUT-FILE
000110 000110 ASSIGN TO UT-S-TAPEIN,
000120 000120 SELECT OUTPUT-FILE
000130 000130 ASSIGN TO UT-S-TAPOUT,
000140 000140 DATA DIVISION,
000150 000150 FILE SECTION,
000160 000160 FD INPUT-FILE
000170 000170 RECORDING MODE F
000180 000180 BLOCK 10 RECORDS
000190 000190 LABEL RECORDS STANDARD
000200 000200 DATA RECORD IS INPUT-RECORD,
000210 000210 01 INPUT-RECORD,
000220 000220 03 FILLER
000230 000230 03 POCL-TYPE
000240 000240 03 FILLER
000250 000250 03 RISK-CODE
000260 000260 03 FILLER
000270 000270 03 ITEM-CODE
000280 000280 03 FILLER
000290 000290 FD OUTPUT-FILE
000300 000300 RECORDING MODE F
000310 000310 BLOCK 10 RECORDS
000320 000320 LABEL RECORDS STANDARD
000330 000330 DATA RECORD IS INPUT-RECORD,
000340 000340 01 DATA RECORD,
000350 000350 03 EXPANDED-RECORD
000360 000360 03 SORT-KEY
000370 000370 PROCEDURE DIVISION,
000380 000380 INITIALIZE SECTION,
000390 000390 START-PROGRAM,
000400 000400 OPEN INPUT-FILE
000410 000410 OUTPUT-FILE,
000420 000420 SET-SORT-KEYS SECTION,
000430 000430 D9999N,
000440 000440 NOTE DECIBLE SET-SORT-KEYS
000450 000450 X 07
000460 000460 CXPOOL-TYPE = 'N' 'S' 'T' 'T' 'T' 'T'
000470 000470 CXRISK-CODE = 'A' 'B' 'C' 'C' 'C' 'C'
000480 000480 C ITEM-CODE = 15 OR 20 OR 30
000490 000490 S READ INPUT-FILE
000500 000500 AT END GO TO END-JCB,
000510 000510
000520 000520 A MOVE 'A' TO RISK-CODE
000530 000530
000540 000540 AXCCMPUTE SORT-KEY =
000550 000550

```

ICS DECIBLE III VER 3 GEN 4

000540	000560		1	2	1	3	4	5	0
000550	000570	A MOVE INPUT-RECORD TO EXPANDED-RECORD							
000560	000580	WRITE OUTPUT-RECORD							
000570	000590		X	X	X	X	X	X	X
000580	000600	A RE-ENTER	X	X	X	X	X	X	X
	000610	END.							
	000620	D99995.							
	000630	READ INPUT-FILE							
	000640	AT END GO TO END-JCB.							
	000650	D9999.							
	000660	IF POLL-TYPE =							
	000670	'N'							
	000680	IF RISK-CODE =							
	000690	'A'							
	000700	GO TO D999951							
	000710	ELSE							
	000720	IF RISK-CODE =							
	000730	'B'							
	000740	GO TO D999952							
	000750	ELSE							
	000760	GO TO D999957.							
	000770	IF POLL-TYPE =							
	000780	'S'							
	000790	GO TO D999953.							
	000800	IF POLL-TYPE =							
	000810	'T'							
	000820	IF RISK-CODE =							
	000830	'A'							
	000840	GO TO D999954							
	000850	ELSE							
	000860	IF RISK-CODE =							
	000870	'C'							
	000880	IF ITEM-CODE = 15 OR 20 OR 30							
	000890	GO TO D999955							
	000900	ELSE							
	000910	GO TO D999956.							
	000920	GO TO D999957.							
	000930	D999953.							
	000940	MOVE 'A' TO RISK-CODE.							
	000950	D999951.							
	000960	COMPUTE SORT-KEY =							
	000970	1.							
	000980	D999958.							
	000990	MOVE INPUT-RECORD TO EXPANDED-RECORD							
	001000	WRITE OUTPUT-RECORD.							
	001010	GO TO D99995.							
	001020	D999952.							
	001030	COMPUTE SORT-KEY =							
	001040	2.							
	001050	GO TO D999958.							
	001060	D999954.							
	001070	COMPUTE SORT-KEY =							
	001080	3.							
	001090	GO TO D999952.							
	001100	D999955.							

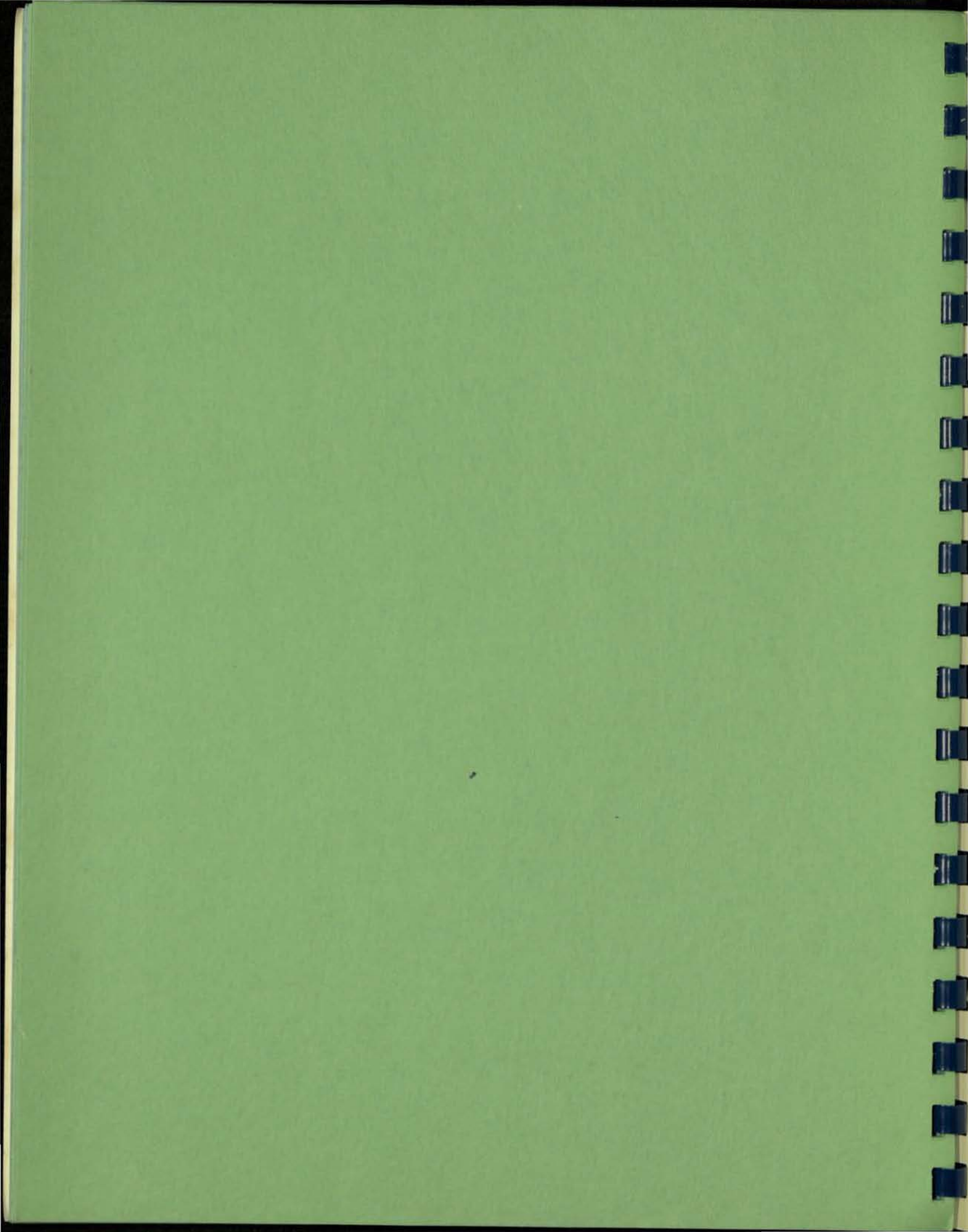
C-4



I C S   D E C I B L E   I I I   V E R 3   G E N 4

```
001110        COMPUTE SORT-KEY =
001120        4.
001130        GO TO D999958.
001140 D999956.
001150        COMPUTE SORT-KEY =
001160        5.
001170        GO TO D999958.
001180 D999957.
001190        COMPUTE SORT-KEY =
001200        0.
001210        GO TO D999958.
000600        001220 FINISH-JOB.
000610        001230 CLOSE INPUT-FILE WITH LOCK.
000620        001240 CLOSE OUTPUT-FILE.
000630        001250 STOP RUN.
```

```
***** 16 WARNING    0 ERRORS    0 WARNINGS    1 TABLES    *****
***** 16 WARNING    0 ERRORS    0 WARNINGS    1 TABLES    *****
***** 16 WARNING    0 ERRORS    0 WARNINGS    1 TABLES    *****
***** 16 WARNING    0 ERRORS    0 WARNINGS    1 TABLES    *****
***** 16 WARNING    0 ERRORS    0 WARNINGS    1 TABLES    *****
```



Copyright, 1971 by Independence Computing & Software Corporation. All rights reserved. Copies of this manual, in whole or in part, in any form, are not authorized without written approval. Additional copies may be ordered from Independence Computing & Software Corporation.



INDEPENDENCE COMPUTING & SOFTWARE CORPORATION,  
235 WHITE HORSE PIKE, WEST COLLINGSWOOD, NEW JERSEY 08107

PROCEEDINGS OF THE  
DECISION TABLES SYMPOSIUM

FOREWORD

This document contains the proceedings of a Symposium on Decision Tables presented September 20, 1962 in New York City. The technical content is the result of extensive work by the Systems Group of CODASYL, chaired by Mr. Les Calkins. The Symposium was co-sponsored by the Systems Group and by the Joint Users Group. The Systems Group was responsible for the technical content and presentation of the material. The Joint Users Group was responsible for arrangements, including reservations and publication of these proceedings.

The Joint Users Group (JUG) is a special committee of the Association for Computing Machinery. It was officially accepted by the ACM in May 1961. JUG is composed of representatives of other computer user groups. It is intended to be a catalyst for action on problems common to several user groups. Its areas of interest are:

1. Common programming languages and other means of communication between computing machines.
2. Establishment and maintenance of standards for communication and distribution of computer programs and techniques.
3. Exchange of information on problems arising from the operation of a computer installation.
4. Communication of methods and techniques for comparing the effectiveness of computer problem solving techniques.
5. Consideration of hardware standards in cooperation with other interested agencies.

At present the following user groups are formal members of JUG:

EXCHANGE  
SHARE  
GET  
FBUG

UUA  
POOL  
RUG  
USE

ACKNOWLEDGMENTS

Systems Group of CODASYL

Les W. Calkins, Chairman of CODASYL - United States Steel Corporation  
Lynn Brown - Insurance Company of North America  
Orren Y. Evans - International Business Machines Corporation  
John J. Feldman - The Howard Savings Institute  
Burt Grad - International Business Machines Corporation  
Howard T. Hallowell, III - Standard Pressed Steel Company  
Mary K. Hawes - Radio Corporation of America  
Charles Katz - General Electric Company  
Sol Pollack - The RAND Corporation  
John R. Smith - Hunt Foods and Industries, Incorporated  
Malcolm D. Smith - Remington Rand  
Jack A. Strong - Computer Sciences  
Richard J. Sullivan - General Motors Research Laboratories  
Richard E. Utman - Remington Rand Univac Division  
Kendall Wright - The Church of Jesus Christ of Latter-Day Saints

Joint Users Group

Leonard V. Parent, Chairman of the Meetings Committee - Trunkline  
Gas Company  
Burt L. Neff - Metropolitan Life Insurance Company  
William Smith - Lehigh University  
Jerry L. Koory, Chairman of JUG - System Development Corporation

TABLE OF CONTENTS

	<u>Page</u>
Decision Tables Symposium Joseph Cunningham, Associate Director of Data Automation, United States Air Force	7
Place of Decision Tables and DETAB-X Les W. Calkins, United States Steel Corporation	9
The Need for Precise Problem Definition Mary K. Hawes, Radio Corporation of America	13
Structure and Concept of Decision Tables Burton Grad, International Business Machines Corporation	19
What Is DETAB-X? Solomon L. Pollack, The RAND Corporation	29
Approaches to Decision Table Processors Kendall R. Wright, The Church of Jesus Christ of Latter-Day Saints	41
Question and Answer Period, Morning of September 20, 1962	45
Commercial and Engineering Applications of Decision Tables H. N. Cantrell, General Electric Corporation	55
Application of Decision Tables to Management Information Systems Frederick Naramore, The Sutherland Company	63
Decision Table Experience on a File Maintenance System Lynn M. Brown, Insurance Company of North America	75
FORTAB: A Decision Table Language for Scientific Computing Applications G. W. Armerding, The RAND Corporation	81
Manufacturing Applications of Decision Structure Tables T. F. Kavanagh, General Electric Corporation	89
Question and Answer Period, Afternoon of September 20, 1962	99



## DECISION TABLES SYMPOSIUM

Joseph Cunningham

I welcome and appreciate the opportunity to participate with you in the Symposium for the next two days. We in CODASYL hope that it will be a milestone in our mutual quest to improve the contribution that we in the data processing community can make by progressively improving, or working toward the improvement of those techniques which we employ, and by introducing new concepts which offer potential for the future.

If we re-live the past few years, to bring us up to the point where we are today, you will recall that the Conference on Data Systems Language was convened in Washington in May of 1959, to consider the programming language problems then existing. It was observed that the community recognized that as long as there was a direct relationship between an individual piece of hardware, and the software accompanying it, we would have a deterrent toward capitalizing on hardware advances. The Conference realized that the individual attempts to develop independent programming languages to solve this problem were resulting in a second order, or generation, of the captivity nature of the situation then existing.

The Conference concluded that the solution to the problem was to develop a language free from any direct hardware relationship, and that the actions which were required, and their sequence, was as follows:

1. To provide a language which was "problem-oriented, but machine-independent." By adopting such a language, or developing one, we would be in a position to accept new hardware improvements as they came along without sacrificing the programming investment which had accrued over the years. This was the immediate, or short-range, task in which time was of maximum importance.
2. A second step was to express, or find a language for expressing, management systems in such a way that they were "systems-oriented and computer-independent." By such a language we would have the flexibility of expressing systems in a manner which would permit implementation, either mechanically, by people, or by computer.
3. A third and more distant step was the attempt to interrelate business system languages with those used in expressing the scientific or computational type requirements. To express this in more current terms, the possible integration of what we now know as COBOL, with the ALGOL or FORTRAN languages.

Through the efforts of a wide number of people (whose names I would like to mention but will not because of the time involved and the probability that I would overlook one or more of the participants and contributors) the language which we now know as COBOL was developed and is at this point reaching the stage where its effectiveness can be evaluated, since compilers have been, and are now being, developed. One of the pitfalls of evaluation is the distinction between the language efficiency and the efficiency of the compiler - it is all too easy to judge the language on the basis of the compiler performance.

During the period of development of COBOL, the CODASYL Executive Committee recognized that it was impossible to accommodate all of the competence and talent which was available within the data processing community. The Short Range Committee, working against time, contained a good cross-section of the data processing community. To the best of its ability the Executive Committee frequently requested that individuals communicate with the Committee any ideas or proposals which would benefit the language. The attempts to secure input from the community generally produced very little constructive suggestions. Time did not permit going to the community in sessions such as we have here today and tomorrow.

Concurrently, the Development Committee of CODASYL under the able direction initially of Mr. Bob Curry, and subsequently and presently of Dr. Arnold Hestenes, has worked actively in the second and third phases which I previously mentioned. The work of examining the possibilities of facilitating the expression of management systems in such a manner as to make them useful for computers and computer programming, but also for other methods of accomplishing the data processing needs, has been underway. It has reached a point in one area where it is now appropriate to present to you one of their recommendations - the Decision Tables. This is being done today and tomorrow through this Symposium.

It was the intention of the Development Committee - Les Calkins and Jack Strong, in particular - that this Symposium would be a vehicle whereby the entire data processing community would be invited to listen to an explanation of the product, to learn how it was intended to be used, to consider it, and as a result provide to the Maintenance Committee the feedback so necessary to their continued endeavors.

We are grateful to JUG for their willingness to underwrite this financially, and to you for your expression of interest, demonstrated by your attendance here. We sincerely hope that this session will be mutually profitable.

## PLACE OF DECISION TABLES AND DETAB-X

L. W. Calkins

Good morning ladies and gentlemen!

I want to take this opportunity to welcome you to this Symposium on Decision Tables and the introduction of DETAB-X for your consideration.

I feel that it is extremely important to provide you with some background before proceeding to the actual agenda topic for which I am responsible. The background will be brief, and yet I hope by your exposure to it that a foundation can be established upon which DETAB-X can be placed in its proper perspective.

On May 28 and 29, 1959, a meeting was called in the Pentagon for the purpose of considering both the desirability and feasibility of establishing a common language for the programming of electronic computers in business-type data processing. Representatives from users, Government installations, computer manufacturers and other interested parties were present. With almost unanimous agreement, the CODASYL effort was formed. In the interest of brevity and with the risk of criticism for interpretation, a Task Force was created whose job it was to submit specifications for a "Machine Independent, Procedure Oriented Language." Their efforts resulted in the publication of COBOL in April, 1960. Without delving into the details, the Executive Committee of CODASYL has established the necessary groups to provide for the updating and maintenance of COBOL.

Within the CODASYL effort is a group known as the Development Committee. The Development Committee has been subdivided into two groups known as the Systems Group and the Language Structures Group. The charge given to the Development Committee was to provide the specifications for a "Machine Independent, Systems Oriented Language." Note here the difference between the two assignments; ours is "Systems Oriented" while that of COBOL was directed to "Procedure Oriented." For me at this point in time to define clearly for you just what is meant by the term "Systems Oriented" is premature; or more honestly stated, I am just not sure of its definition. We as a committee are sure of one thing, however, and that is that there is a great deal more that must be learned before we can truly approach the task given us. In this light then, I believe it will become apparent why we are presenting this Symposium on DETAB-X to you.

The Systems Group has devoted a considerable amount of time to date in discussing possible approaches to solving the task given it. We have invited and received many outside presentations covering a wide latitude of techniques associated with various concepts. We have chosen to explore the tabular

format. This choice, however, must be qualified for you, in that it represents a path of experimentation and it should not be considered a final selection.

What then is the place of Decision Tables and DETAB-X? What are some of the attributes of this methodology that have led us to experiment with it?

First of all, a Decision Table is a way, and more particularly, an organized way of expressing the logical decisions that must be made, or that are involved, in a given problem or system, and the resulting actions to be taken based upon those decisions. We believe this approach to be convenient in areas where the logic is complex. We expect this convenience to persist to lower levels of complexity; however, the degree to which this will be true is dependent upon actual use and the resultant evaluation of the experience derived. Permit me to interject at this point that we hope to determine this, among many things, from the feedback from you in using this method.

Reflecting a moment upon our given task--that is, "A Systems Oriented, Machine Independent Language"-- this method offers the possibility of being an effective tool for the systems man in the area of problem analysis; and with the inherent flexibility in setting it up, the connotation could be made that it is systems oriented.

Further, I think we will all agree that a fairly large percentage of the time between the decision to mechanize and the actual production running of an application is devoted to problem definition. We have had indications that this methodology can materially reduce this time. In this light, it can, if proven, be an effective means of reducing cost.

It would be possible to continue this conjecture at great length regarding the possibilities which we in the Systems Group can envision. It is only fair, however, to leave the validity of such conjecture to a sound evaluation of the comments and factual information received and compiled from users of the method.

Now let us turn to DETAB-Z specifically and attempt to place it in the proper perspective. First, let me dispel any idea in your mind about this as being a language. Today we have more languages than we know what to do with. Perhaps I can draw an analogy for you which will clearly set forth our position in this regard. I heard a story the other day concerning two Israelites who ventured into Arab country and were promptly captured. Of course a speedy trial was conducted and the two were sentenced to die before a firing squad. The day of execution dawned and the two were led to a courtyard and placed with their backs to a wall. An Arab approached the first Israelite and asked him if he wanted a blindfold, whereupon the Israelite spit in his eye. His companion turned to him with sadness in his eyes and said, "Meyer, why are you always trying to make trouble?"

We are therefore not here to present DETAB-X as a language and to cause any trouble in that regard. We have in fact used COBOL as the language insofar as we could. This in my opinion was for convenience. First, the operators were available and defined. Second, the work currently being done by the various manufacturers on COBOL compilers would perhaps ease the problem for them in producing machine language from DETAB-X. So clearly then, the objective is to have you experiment with Decision Tables per se, as defined by DETAB-X.

In the interest of clearing up any possible misunderstanding and in the light of wanting to put to rest once and for all any rumors you may have heard, let me emphatically state that DETAB-X is not intended in any way to replace COBOL '61. Again this emphasizes, at the risk of being redundant, that we want you to experiment with DETAB-X in the context of the Decision Table methodology.

Now let me tell you of our objectives as they relate to the feedback we want from you in using DETAB-X.

First, would Decision Table format be useful as an additional form for the Procedure Division of COBOL '61? Let me say here that I want to emphasize the word "additional" and this should not, at this point in time, be misunderstood to imply replacement.

Second, would Decision Table format be useful for problem analysis and within what range of complexity is it effective? In this regard, we hope, by the careful consideration of your opinions, to determine the weak points of the method and take corrective steps wherever possible.

Third, does it act as an effective tool in the area of man-to-man communication and solution documentation? This particular area is one that has plagued us all since the very beginning of mechanization.

Fourth, and in conclusion, would Decision Table format be valuable in an advanced systems-oriented language? This perhaps will be the most complex consideration of all. Of all of our objectives, this one can only be resolved by the most serious and careful discussions possible. If your feedback and our discussions confirm this point, then this will materially affect the future work in the development of a "Systems Oriented, Machine Independent Language" by the Development Committee.

I trust that what I have told you in my remarks this morning has in some way laid the groundwork for the presentations that are to follow. I further hope that I have provided a framework and attitude within which you will accept DETAB-X for what it is.

In any research effort, there is a time when you can no longer afford to keep your work within the laboratory. You must expose your efforts to the reality

of those who could be the ultimate user and be guided by his reaction and constructive criticism. In this light, we are asking for your help by using DETAB-X to aid us in reaching our objectives.

So that we may effectively coordinate your responses, we have established a focal point of contact through which you may send your comments either written or verbal as well as examples of your work. The point of contact is as follows:

Mr. Sol Pollack, The RAND Corporation, 1700 Main Street, Santa Monica, California

This will insure that the Systems Group will receive all feedback in an orderly way.

In closing, I hope you will find the presentations both interesting and informative and that you will apply DETAB-X upon your return home. Thank you.

## THE NEED FOR PRECISE PROBLEM DEFINITION

Mary K. Hawes

SUMMARY

The need for precise problem definition is one of the greatest facing the users of electronic computer systems today. Experience indicates over 65% of the costs associated with programming data processing problems can be attributed to this area. Looking ahead to real-time information processing systems, the need becomes even greater and, furthermore, must be handled at the systems level.

Introduction

Management has asked that a method be developed for defining information processing type problems at a systems level so that management can be assured (1) that the problem has been completely defined (2) that its implementation using electronic computer systems will cost in both time and money the approximate amounts projected, (3) that the production cycle meet the specifications, and (4) that there be the ability for modifying the problem in accordance with the dictates of the dynamic nature of our world of today. I question if there are any in the audience who do not have some reservation regarding the possibility of attaining this goal because of experience with information type problems. However, I am also certain that, at the same time, the management persons among you are aware that it is a goal that must be met--it is from you that this challenging task has come with the full knowledge that it will take the concerted effort of many to develop and refine techniques for defining information processing type problems at the systems level.

In listing management's requirements in terms of (1) problem definition, (2) computerized implementation, (3) meeting of production requirements, and (4) ability to modify, the need for precise and complete problem definition was placed first because it is perhaps both the most important and the most difficult to achieve, especially in light of the influence it has on the remaining three areas. Another aspect of problem definition that is an integral part of the other areas and which might well be thought of as a separate requirement for emphasis, is adequate documentation. The lack of adequate documentation that exists today regarding data processing systems is deplorable and in some cases is being recognized as a situation that can lead to legal action. It is not to be inferred that the application of electronic computers to problems of business and industry has resulted in the creation of such a situation; but, rather, their application has resulted in discovering the existence of questionable procedures and data errors of which there was little knowledge. Problems that have been put on electronic computer systems are far better defined,

controlled, and more error free than ever before. In fact, this improved control can be considered to be of greater importance to business and industry than the increased rate of processing that is achieved through the use of computers. This improved control of which management has had a taste, is directly responsible for management's request for complete problem definition at the systems level and for the noticeable trend in planning for real-time information processing systems as opposed to the more commonplace serial data processing systems of today.

### Systems-Level Language

There are some of us today who feel this need for precise problem definition can be achieved through the development of a systems-level language for information processing. The question then arises as to what is meant by a systems-level language. A systems-level language for information processing can mean a method for describing a large and complex information processing problem in terms of a rigorous language with well defined rules so that such descriptions can be fed into a computer for processing which will result in the detailed design of the required information processing system together with the required machine instructions, operating procedures and adequate documentation. It is anticipated that such a problem description might be in terms of output requirements, input data, relationships of inputs to outputs, time requirements, system constraints, management policies, and other related facts which can be tagged as environment.

### Costs of Problem Definition

In attempting to stress the importance of precise problem definition in a very limited amount of time, I shall summarize in rather general terms our experience to date on the relative costs associated with problem definition and then spend the major amount of time projecting the type of information processing systems we are contemplating and the implications therein for problem definition.

Cost analysis studies carried on by many users of electronic computer systems indicate that programming costs are approximately equivalent to those of the computer system on which the problem is to be processed. Programming is defined as including the statement and definition of the problem (which by its very nature includes at least a tentative solution in fair detail), the coding of the problem in some language acceptable to a computer system, the debugging of the problem, and the documentation of the problem with its solution so that changes can be made to the problem as such become necessary. Experience indicates that the cost of defining the problem accounts for 65 to 75 percent of the programming costs. Many of the time delays, recoding and extra debugging can also be attributed to inadequate problem definition. The costs of documentation have seldom been excessive due to redocumentation resulting from inadequate problem definition; excessive costs can, however, be attributed to the lack of documentation.



### Impact of Problem-Oriented Languages

Great strides have been made during the past few years to develop programming languages which are oriented to the language of the user. COBOL, ALGOL, JOVIAL, and IPLV are examples of such languages. In each of the above languages, procedural statements are written as inputs to the language system from which machine instructions are generated. The order of the procedural statements prescribe explicitly the sequence of machine instructions as they will finally appear and also the exact order of the processing which will be done. These languages, together with their supporting programs, have helped in cutting down the coding, debugging and documentation time. Some of the error analysis routines may also have contributed to decreasing the problem definition costs. However, we find ourselves without any generally accepted standard tools and techniques for the description of a problem at the systems level that will assist in the design of an information processing system. It is this void which we hope to partially fill through decision table techniques.

Both COBOL and JOVIAL have information processing system implications. Some of the basic concepts of an operating data processing system are reflected in the file and data descriptions of COBOL together with the implied Input/Output Control System. It is possible to do some file organization experimentation on a trial and error basis using COBOL; in fact, some COBOL compiling systems are beginning to incorporate file design criteria which reflect the results of their experience. Any of the programming languages that lend themselves to automatic segmentation of the implemented programs encourage a modular concept. The procedure module concept is not new in that a subroutine for computing the sine of an angle is a procedure module. The same concept has been extended to cover many data processing procedures such as Match, Update, SORT, FILE and LIST. Other data processing procedures that reflect management policies can also be developed as modules, such as Vacation Due, Economical Size Lots, Reorder Point, and Credit Standing.

### Some Aspects of a Systems-Level Language

At the present time we have no programming language that translates some statement of the problem made in terms of the entire system into the procedural statements that are required for a solution to the problem. At the present time, programming languages handle each run as a separate problem. The programmer is entirely responsible for designing the entire system into a series of runs. The programming system assumes the procedural statements as written by the programmer reflect the proper usage of the specified equipment. One of the major reasons for wanting a systems-level language is to make it possible to automatically generate the required procedural statements and then segment such into optimum "bites" consistent with good systems analysis and proper usage of equipment for a smooth operating system.

Considerable experience has been accumulated on automatic segmentation when the procedural statements have been packaged into procedure modules. So the major problem confronting us is that of developing a method for defining the systems-level problem from which we can generate the specific procedural modules required. An added refinement would be to include extensive systems analysis that would evaluate different procedural approaches, different organization of files information together with different equipment configurations in order to generate an optimally designed system.

An added requirement, that of the up-to-the-minute information module, must be added to the above when we recognize the real-time information processing systems that are currently being planned. Such systems are the natural outgrowth of the dynamic nature of our business and scientific environment. Such systems will incorporate communication networks and most processing will be done without human intervention from the time a transaction enters the system, an inquiry is made, or a special report is requested until the results have been furnished.

Relatively few of us have had experience with real-time information processing systems. It might be well to project some of the aspects of such systems in order to realize how even more important it will be to define our problems completely and precisely.

#### Real-Time Information System Requirements

The first prerequisite of real-time information processing systems is to have available or easily accessible, all file information that is likely to be used along with the procedures for identifying, verifying and processing the input information. As an accompanying requirement, various records must reflect the above actions for purposes of subsequent actions, audit trails, safety of records, management information and various statistical studies of a dynamic or postoperative nature. Such systems will require millions or billions of characters in mass storage. Here mass storage is defined as auxiliary storage to the computer main memory and implies relatively short access time where accesses can be in a random order with respect to the last item accessed. It is interesting to realize what such mass storage means in terms of computer programs; for example, if 20 billion characters of storage are required for data storage, what about an additional million or ten million characters for programs and program control? This in itself results in different approaches to programming techniques and system control.

Having mass storage available need not necessarily imply that all processing will be of a real-time nature. It is quite probable that most large systems will use batch processing to a considerable extent because of time and cost considerations. This projection is based on the assumption that considerable time will be saved by packaging similar types of input, even if the procedure modules used are the same ones which would have been used had real-time been

employed. It is contemplated that partial processing will be employed to determine which inputs are to be processed immediately, which are to be stored temporarily to be processed later, or to be batch processed at some scheduled hour. It is also contemplated that in some cases of batch processing, verification procedures may be processed in advance of the remaining procedures so that action can be initiated if some clarification is needed with regard to the input.

The greatest difference will come in the organization of the information itself together with a parallel mode of operation as opposed to a serial mode. By this latter statement is meant that the changes to the information content of a "master record," brought about as a result of a given transaction, will be reflected in all affected areas essentially in parallel. In serial processing, the effect of a group of transactions is applied to the given files of similar information, and at subsequent intervals of time, reprocessing is performed on those files for various reports and studies. A static file is kept on all stages of development. When using dynamic mass storage, there will be many semi-complete modules of information much like subassemblies in a manufacturing operation. Such information modules will also be assembled much as the subassemblies in that they will be put together in some prescribed manner, with or without additional processing or finishing. They differ significantly from the manufacturing process which has been used as an analogy, in that the information module having been used in a prescribed manner for a specific output is still available for use as many times as it appears to be necessary.

Another major difference anticipated that will have an impact on audit procedures and accuracy checks in general, is that there will be less moving of data from one place to another to indicate a given process has been performed. The completion of the process will frequently be indicated by changing the label associated with the data. The actual physical location of both data and instructions will be less significant and need only be known by the processor. Extra care will have to be exercised with the labels since unlabeled or incorrectly labeled data will be difficult to locate or to identify in case it is found.

We are familiar with the master-file record information module; however, there are others which must be available at all times. These are similar to the intermediate results that are being developed and accumulated during various runs in our current serial type procedures. An example of such are the various figures being accumulated for sales analysis, market projections, rate of change in inventories, etc. Such information modules should be kept at the lowest meaningful level to balance costs of storage and costs of processing. It should be kept in mind when developing a given system that once digital values have been combined, the identity of the individual elements are usually lost and cannot be obtained by fracturing the resultant value. It will be necessary to anticipate the types of special reports management will be requiring in order to have the appropriate information modules in a semi-finished form.

There will be insufficient time to generate this type information for most reports requested in an unscheduled manner.

Let us return for a few minutes to the procedure modules and realize the implications on real-time information processing systems. An easy way to visualize the situation is to assume that all the runs for a given data processing system are suddenly only one run and that all instructions are either in main memory or can easily be brought into main memory as required. The procedural module concept is extremely important in that a given module used in a number of different runs can now probably exist in only one place. Furthermore, once a given input has been identified, the procedure modules which are to be used in processing will probably be identified by an associated pattern. This will be somewhat different from the serial processing in which a series of conditions and actions are all intermixed so that a given transaction may not be fully defined until after it has been completely processed.

The task of incorporating changes to the system takes on added color in that these changes must take place dynamically. Special techniques must be developed to assure no damage is done when programming errors do occur. Complete logs must also be kept and be reflected in such management reports as may be involved. It is anticipated that most changes in such a system would be in the nature of additions or deletions at a module level. When changes are incorporated to reflect an improved procedure, it should be possible to modify the appropriate procedure module or modules without too much trouble in that modules are self-contained. It will be absolutely necessary to set up tight controls so that unauthorized changes cannot be made. Complete documentation is an absolute requirement and, therefore, should be done automatically to assure its being accomplished in an understandable manner.

#### Conclusion

Much work is yet to be done in order to specify how to define such a system in detail with the degree of preciseness that is required. Still more work will be required to test and modify the various system-level language developments. We do feel that Decision Tables will be most useful and that by experimenting with real live problems, we should arrive at tentative definitions and techniques that will lend themselves to improvement as we learn more about designing and controlling real-time information processing systems.

## STRUCTURE AND CONCEPT OF DECISION TABLES

Burton Grad

People are different. Some prefer foreign compact cards; other want roomy domestic models. Some people like chocolate ice cream; others favor strawberry. Differences in taste and preference are personal considerations which make life more interesting and encourage industry to turn out a continuous stream of new and unique products.

Individual differences also appear in the several philosophies and approaches to designing business systems. Some look at all business problems as an exercise in file maintenance or information retrieval. Others see the same problems in the framework of mathematics or arithmetic statements. There is another school which views problems in terms of input transformation or output preparation. Still others are principally concerned with the procedures and operational sequences.

While each of these individual approaches is perfectly valid in certain situations, none of them can be recommended as a universal technique for solving all business systems problems. The reason for this is that the systems design must cope with such a large variety of problems that different techniques are needed for maximum efficiency. For example, it wouldn't seem reasonable to use the same rod and reel for trout fishing as you would use to take marlin. As the classes of problems handled by computers become increasingly more complex, new and improved tools and techniques are needed to solve them.

Decision Tables, a recent development, provide a means of presenting complex decision logic in a way that is relatively easy to prepare and understand. A decision table shows the specific alternative courses of action to be taken under various combinations of conditions. This permits an analyst or programmer to concisely and completely record logical decision rules for analysis, documentation and programming. By discussing decision table structure and concept, you will begin to see why decision tables may soon become another important tool for systems design.

BASIC STRUCTURE

The basic outline of a decision table indicates the four most significant quadrants (see figure 1). Conditions are shown above the horizontal heavy line. The condition stub contains the common condition information. Figures or words which supply a concrete value or range of values to the condition named are shown in the upper right-hand, or the condition entry area. Names or titles of resulting actions are written in the action stub, in the lower left portion of the table. Related values and ranges of values for each action named are located in the lower right quadrant.

Special identification information common to the entire decision table is entered into a table header, which includes table number or name, sequence control instructions, and the number of conditions and actions in the table. Data common to the several rules in a table are placed in a rule header. This might contain rule number and frequency of occurrence.

When all the conditions in a single entry column of a table are satisfied, then all actions shown directly below are executed. This combination of conditions and actions is called a decision rule.

### INSURANCE TABLE

These seven terms allow us to reference the various areas of a table. Let's examine a sample insurance premium decision table with four independent and three dependent factors (see figure 2).

Conditions for issuing a policy are stated as Health, Age, Section of Country, and Sex; these names are shown in the condition stub. Each condition name has a series of values, which are arrayed in the condition entry area. Thus, health can be good or poor; section of country can be east or west.

In a like manner, names of actions are listed in the action stub for Premium Rate, Policy Limit, and Type of Policy.

Rule one of the insurance table reads:

"IF a person's health is excellent, and the person is between 25 and 35 years of age, and lives in the eastern section of the country, and is of the male sex, THEN his premium rate is \$1.27 per thousand, and his policy may not be written for more than \$200,000, and he is issued policy type A."

The word IF prefaces a list of conditions; the word THEN is used to show transitions from conditions to actions. The connective AND is always used to indicate the relation of one condition to another, or the sequence of one action to another.

Turning back to the insurance table, the last rule reads:

"IF a person's health is poor, and age is 65 or over, and the person lives in the West, and is female, THEN the policy rate is \$9.65 per thousand, and the policy limit is \$10,000, and a type B policy is issued."

In reading these two rules, individual condition and action information for each row is extended from the stub to the corresponding entry row of the table,

e.g., "If sex is male; if sex is female." This characteristic establishes the fact that this is an extended entry table.

In this table, only one set of actions can take place since only one set of conditions can be satisfied at a time, i.e., there is only one successful rule per pass through the table.

#### CREDIT TABLE

Having considered the basic table terminology and its application with an extended entry table, another type of table can now be introduced.

For example, this credit table (figure 3) while similar to the insurance table, has a number of different properties. Rule one reads:

"IF credit limit is OK, THEN approve the order."

And the second rule would be read:

"IF the credit limit is not OK, and pay experience is favorable, THEN approve the order."

This is a Limited Entry Table, and differs from extended entry form in that the entire condition or action is written in the stub area of the table. Notation in the condition entry area of the table is limited to indicating whether a particular condition should be asserted (Y), negated (N), or ignored altogether (blank or dot). In action entries, an X indicates that the corresponding action should be executed, while a blank means that it should be ignored.

A table which includes both extended and limited entry rows is called a Mixed Entry Table.

Use of limited entry format permits including more rules than is physically possible with extended entry. But excessive use of limited entry tends to extend a table vertically. A balance between these two is achieved by employing the mixed entry form. The prevailing conditions an analyst meets in a study often help decide which form to use.

An unconditional table is composed of one or more actions, but no conditions. As is the practice in all tables, the actions are executed in the order they are written.

Two major types of sequence control can be exerted through an action row command:

1. When an action row is reached, control can be temporarily transferred to another referenced table, and when that table has been processed, control will revert back to the succeeding action in the original table.

2. When the last action in a rule is reached, control can be directed to a new table.

A table may be entered at only one point, although there may be as many exits from a table as there are decision rules in it.

Where the exit is always to the same table, no matter which rule is satisfied, then space can be conserved by inserting the sequence control command in the table header for automatic execution after each series of actions has been completed. With this ability to signify temporary or permanent transfer to other tables, a data processing system can be divided into logical segments and structured for effective problem analysis.

#### FILE UPDATE TABLE

Up to this point we have examined tables which were already prepared. Now let's start from scratch and build up a table for a typical file maintenance problem. The basic problem elements are: a master file and a detail file serve as inputs; a new master file and an error file are produced as outputs. Within the computer, three basic areas are assigned: master, detail, and new master. The task is to determine and record the logic by which the incoming master file is modified from information in the detail file to prepare a new and updated master file containing any changes and additions, and from which deleted records have been eliminated.

The rules in figure 4 will be considered sequentially. What are the appropriate conditions and actions for the starting situation? This requires a single condition, start, and actions for reading one master card and one detail record card into their corresponding memory areas. The final action returns us to the beginning of this table.

We will need a rule to handle an end-of-job condition when the end-of-detail and end-of-master are reached. Therefore, we add two conditions to the condition area of the table under rule 2, and also indicate in the last action row a transfer of control to a closing routine which provides for sentinels, tape marks and so forth.

Next, we must consider what happens when the end-of-detail is reached, but not end-of-master. Since there can be no further changes, additions, or deletions to the original master when this occurs, we need one new action in rule 3: write the updated master from the master area. Then we read another master and return to the beginning of the table.

In rule 4, we want to take care of the condition where end-of-master has been found, but not end-of-detail. Consequently, the remaining details will be additions to the master. This is signified by a new condition row and two new action rows. The addition switch is set "ON." The information in the detail



area is moved to the new master area; a new detail record is read; and sequence control is transferred to another table identified as CHANGE.

The next three rules are concerned with cases where neither the master nor the detail file has been completed, and an identification number must be compared between the two records.

Rule 5 considers the case when the detail identification number is less than that of the master, and, therefore, the logic of rule 4 should be followed. In rule 6, the detail is greater than the master, and the logic expressed in rule 3 applies.

Rule 7 covers the case where the master identification number is the same as that of the detail record; information in the master area is moved to the new master area, with control transferred momentarily to the CHANGE table.

The final rule is a special one which will be executed only upon failure to satisfy any other rule. Since all legitimate possibilities for this situation have been explicitly covered, such failure may represent a logical error or invalid data; therefore, an error routine is carried out, another detail record is read, and control returns to the beginning of the table. This ELSE rule will also take care of sequence errors in the master file, any non-matching detail which is not an addition, and certain types of sequence errors in the detail file.

Some tables are written purposely so that the rules do not exhaust all combinations of conditions, and in this situation, the unconditional rule ELSE tells what to do if none of the other rules can be executed.

#### SUMMARY

A decision table is divided into four major areas, separated by heavy (or double) horizontal and vertical lines. Conditions are located above the horizontal line, actions below. Names of values are placed in the stub to the left of the vertical demarcation line; specific values and ranges of values are arrayed in columns of the entry area, to the right.

Conditions and actions have a cause and effect relationship; no actions may appear in the condition area; no conditions can be indicated in the action area.

Information common to the table is written in the table header; information pertinent to each rule is placed in the rule header.

A decision rule is read by proceeding sequentially down a vertical column of the entry area and combining related information from the stub area with its

associated entry to produce a complete statement. Where multiple conditions exist, all conditions must be satisfied before the actions for that rule can be executed.

There can only be a single success for any one pass through a table, i.e., no more than one set of actions can be executed for any given set of input values.

In limited entry table form, the entire condition is written in the stub; the entry area is used to show whether a certain condition is true, false, or not pertinent. If an action is to be performed, it is noted by an X; otherwise the action entry is blank.

Extended entry tables differ from limited entry in that part of the condition or action statement is extended into the entry area. Both types of entry may be shown in a mixed entry table.

One special type of table is the unconditional, or one rule table.

Sequence control within a table requires that action be executed in the order listed. Between tables, sequence control can provide for a temporary switch to another table, or a complete transfer. The table header can also be used for sequence control information.

A table may be entered only at a single entry point, but it may have multiple exit points.

The "all other," or ELSE rule provides an unconditional rule to be used when none of the other rules is satisfied.

In succeeding presentations, these same basic conventions will reappear time and time again in essentially the same form and pattern they have been presented here. While some of these conventions may seem restrictive, they provide a common basis and framework for initial experimentation.

TABLE HEADER	RULE HEADER			
CONDITION STUB	DUC-S-OZ MTCR ZO-S-CMD	CONDITION ENTRY		
ACTION STUB		ACTION ENTRY		

25

Figure 1. Decision Table Structure

HEALTH	EXCELLENT	EXCELLENT	POOR
AGE	$\geq 25, < 35$	$\geq 25, < 35$	$\geq 65$
SECTION OF COUNTRY	EAST	EAST	WEST
SEX	MALE	FEMALE	FEMALE
PREMIUM RATE	1.27	1.18	9.82
POLICY LIMIT	200,000	100,000	10,000
TYPE OF POLICY	A	B	R

Figure 2. Insurance Table

TABLE:CREDIT	RULE 1	RULE 2	RULE 3	RULE 4
CREDIT LIMIT IS QK.	Y	N	N	N
PAY EXPERIENCE IS FAVORABLE		Y	N	N
SPECIAL CLEARANCE IS OBTAINED			Y	N
APPROVE ORDER	X	X	X	
RETURN ORDER TO SALES				X

27

Figure 3. Credit

TABLE: UP DATE	01	02	03	RULE #		06	07	08
				04	05			
START	Y	N	N	N	N	N	N	ELSE
END OF DETAIL		Y	Y	N	N	N	N	
END OF MASTER		Y	N	Y	N	N	N	
DETAIL VS. MASTER					<	>	=	
DETAIL IS AN "ADDITION"				Y	Y			
DO ERROR ROUTINE								X
MOVE MASTER TO NEW MASTER							X	
MOVE DETAIL TO NEW MASTER				X	X			
SET ADDITION SWITCH				ON	ON		OFF	
WRITE MASTER			X			X		
READ MASTER	X		X			X		
READ DETAIL	X			X	X			X
GO TO TABLE	UP- DATE	END	UP- DATE	CHG.	CHG.	UP- DATE	CHG.	UP- DATE

Figure 4. File Up-Date

## WHAT IS DETAB-X?

Solomon L. Pollack\*

Burt Grad's talk has described what decision tables are. I would like to tell you about a specific decision-table language, DETAB-X (Decision Tables, Experimental), an experimental language that combines COBOL-61 and decision tables. It is a proposed supplement to, not a replacement of, COBOL-61.

The CODASYL Systems Group has designated DETAB-X as an experimental language in order to emphasize that it is available on a test basis to those in the business data processing or scientific field who are willing to experiment with it. Hopefully, users of the language will provide feedback concerning its merits and defects to the CODASYL Systems Group.\*\*

Since COBOL-61 is an integral part of DETAB-X, let us turn to the first chart. As most of you know, source programs written in COBOL-61 consist of four major divisions: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE. The chart lists in broad outline what is contained in each division; the COBOL-61 manual provides the detailed specifications.

The specifications for the IDENTIFICATION and ENVIRONMENT DIVISIONS of DETAB-X source programs are exactly as prescribed in the COBOL-61 manual. The DATA and PROCEDURE division specifications, however, differ enough from those of COBOL-61 to require a supplementary manual.\*\*\*

DETAB-X is designed so that source programs written in DETAB-X can be translated by people or a computer preprocessor to COBOL-61, which can then be translated to an object (computer) program by a COBOL compiler (most of which will be available by the end of this year). This is not to preclude

---

\* Any views expressed in this paper are those of the author. They should not be interpreted as reflecting the views of The RAND Corporation or the official opinion or policy of any of its governmental or private research sponsors. Papers are reproduced by The RAND Corporation as a courtesy to members of its staff.

\*\* Criticisms and suggestions concerning DETAB-X should be sent to Sol Pollack, The RAND Corporation, 1700 Main Street, Santa Monica, California.

\*\*\* This supplementary manual, "Preliminary Specifications of DETAB-X," will be distributed to the attendees of this symposium.

## Chart I. Divisions of COBOL-61 Source Program

IDENTIFICATION DIVISION

NAME OF SOURCE PROGRAM  
AUTHOR  
DATE  
REMARKS

ENVIRONMENT DIVISION (EQUIPMENT)

NAME OF COMPUTER  
1) FOR COMPILING SOURCE PROGRAM  
2) FOR RUNNING OBJECT PROGRAM  
MEMORY SIZE  
NUMBER OF TAPE UNITS  
PRINTERS, ETC.

DATA DIVISION

1. FILE SECTION
2. WORKING STORAGE SECTION
3. CONSTANTS SECTION

PROCEDURE DIVISION

SECTIONS AND PARAGRAPHS



enterprising users or manufacturers from writing compilers that will translate DETAB-X source programs directly to computer object programs.

Let us now turn our attention to DATA DIVISION. There is one major difference between the DATA DIVISION specification of DETAB-X and that of COBOL-61. DETAB-X uses a table format for describing data; COBOL-61 uses a free-form English format. To illustrate, on Chart 2 we show some data described according to COBOL-61 specifications. You will note even in this small example a great deal of redundancy. Also, it is very difficult to check that all the attributes of each data item have been specified.

In Chart 3, we have described the same data as in Chart 2, but have used the table structure of DETAB-X. Notice that the headers in this chart eliminate the many redundancies appearing in the COBOL-61 example; we have thereby reduced the amount of writing by the system analyst or programmer. Also, because in the DETAB-X data description we have all data attributes in the heading, the chances of leaving out a necessary attribute of any data are decreased.

From a communications point of view, the system designer will find the people in the applications are more disposed to helping him check the data description if it is in tabular form as contrasted with the free-form style of COBOL-61. The table isn't as cluttered as free-form English style and therefore is much easier to read.

While we have made some improvements in data description for DETAB-X, the big payoff is in the Procedure Division. In this area we use decision tables for describing the many decision rules that exist in business operations. To illustrate the differences between COBOL-61 and DETAB-X in describing procedures, I have extracted an example from Jean Sammet's article on COBOL-61 in the May issue of the ACM Communications and copied it onto Chart 4.

Note that COBOL-61 is serial in nature. The comparisons and the actions based on those comparisons must occur in the order in which they are specified. Note also that this form does not lend itself easily to analysis or to checks for completeness and accuracy. It is difficult to tell whether all the appropriate comparisons on stock-on-hand, current order, and secondary-supply have been made. Also, if a comparison is made against several values, it is very difficult to spot wrong values, because corresponding values appear in different paragraphs, some distance from each other.

Let us turn to Chart 5 which shows these same rules in decision-table form. Notice that having the conditions laid out in tabular form enables the system designer to better determine if he has considered all the possible combinations of conditions that might occur. He knows for example that if there are three conditions that can be satisfied or not satisfied, there is a total of  $2^3$  or 8 different rules he might form.

## Chart 2. Sample COBOL-61 Data Description

01 INVENTORY RECORD: CLASS IS AN  
03 PART-NAME; USAGE IS DISPLAY; PICTURE IS LA(11)  
03 ON-HAND-QUANTITY; USAGE IS COMPUTATIONAL;  
SIZE IS 6  
03 ON-ORDER-QUANTITY; USAGE IS COMPUTATIONAL;  
PICTURE IS 9 (6)  
01 SALES-REPORT; CLASS IS AN  
02 DISTRICT-SALES; OCCURS 103 TIMES  
03 DISTRICT-NUMBER; USAGE IS DISPLAY; PICTURE IS 999  
03 UNIT-SALES; USAGE IS COMPUTATIONAL;  
PICTURE IS 999999V99  
02 TOTAL-SALES; USAGE IS COMPUTATIONAL;  
PICTURE IS 999999V99

Chart 3. Sample DETAB-X Description

Line No.	Level No.	Name	Abbr	Use Code	Desc Type	Pic Ref or Value	Repets	
							Min	Max
001	01	INVENTORY-RECORD	INV-REC					
002	03	PART-NAME		D	P	LA(11)		
003	03	ON-HAND-QUANTITY	OHQ	C	P	9(6)		
004	03	ON-ORDER-QUANTITY	OOQ	C	P	9(6)		
005	01	SALES-REPORT	SAL-REPT					
006	02	DISTRICT-SALES	DI-SALES				103	103
007	03	DISTRICT-NUMBER	DI-NR	D	P	999		
008	03	UNIT-SALES		C	P	9999999V99		
009	02	TOTAL-SALES	TOT-SAL	C	P	9999999V99		

Chart 4. Sample COBOL-61 Procedure\*

IF STOCK-ON-HAND IS LESS THAN CURRENT-ORDER THEN IF CURRENT-ORDER IS GREATER THAN SECONDAY-SUPPLY GO TO EMERGENCY-ORDER-ROUTINE; OTHERWISE PERFORM SECONDAY-SUPPLY-ROUTINE; OTHERWISE SUBTRACT CURRENT-ORDER FROM STOCK-ON-HAND.

Chart 5. Sample DETAB-X Procedure

	Rule 1	Rule 2	Rule 3
STOCK-ON-HAND LR CURRENT-ORDER	Y	Y	N
CURRENT-ORDER GR SECONDARY SUPPLY	Y	N	-
GO TO	TABLE 3	TABLE 4	-
SUBTRACT CURRENT-ORDER FROM	-	-	STOCK-ON-HAND

NOTE: TABLE 3 is an emergency-order routine

TABLE 4 is a seconday-supply routine

\* Borrowed from Jean Sammet's article, "Basic Elements of COBOL-61," in Communications of the ACM, May, 1962.

DETAB-X differs from COBOL-61 in that rules in the table do not have to be executed in the order they have been written, i.e., rule 1 does not have to be executed first. This gives the compiler freedom to determine the order of rule execution based on some parameter such as frequency of occurrence. For example, if a particular rule is executed 90% of the time while the remaining rules are executed only 10%, it is certainly more efficient to have that 90% rule executed first. The format of DETAB-X makes it easy to specify the parameter for each rule so that more efficient object programs can be developed.

When the rules of a table have been specified, the system designer can add a final rule to the table - ELSE GO TO TABLE \_\_\_\_\_. This rule, by definition, is always referred to last, i.e., if the conditions of each of the other rules have not been satisfied. This feature will prove very valuable to business systems. If after a data processing system has become operational, a condition arises that was not anticipated by the system designers, this "else rule" will bring this condition to the attention of the staff. For example, suppose Rule 3 in Chart 5 had been omitted from the table and some time after the system became operational the stock-on-hand was not less than the current-order. Rules 1 and 2 would be tested and found to be unsatisfied. The "else rule" would then be automatically referred to and the computer could print out that specified rules had not been satisfied. Thus an incomplete table could be spotted the first time the missing conditions were met.

To further illustrate the difference between COBOL-61 and DETAB-X, Charts 6 and 7 describe the rules for computing depreciation and lease expenses.

The language used in the decision tables of DETAB-X is a modified COBOL-61. The deviations of DETAB-X from COBOL-61 (deletions and additions) are described in the DETAB-X Specifications Manual and will be discussed in detail at tomorrow's tutorial sessions. Let me again emphasize that source programs written in DETAB-X (using modified COBOL-61) can readily be translated to standard COBOL-61.

One more point. As you have probably inferred from Burt Grad's talk on decision tables, there is little point to forcing a series of unconditional actions into a decision-table structure. DETAB-X therefore allows portions of the Procedures Division to be written in COBOL-61 sections and paragraphs. However, where there are decision rules (sets of actions based on sets of conditions), we strongly recommend that decision table structures be used.

In Chart 8 we have listed some desired goals for future business languages. It is our hope that DETAB-X is a step in this direction. We feel that DETAB-X can help users in documenting their system and that programs written in DETAB-X will provide improved communication between system designers, programmers, and functional specialists. DETAB-X is also expected to increase the accuracy and completeness of problem statement achievable by existing languages. It is available to anyone willing to try it and the Development Committee would appreciate receiving any information on the merits and defects of the language.

Chart 6. Sample COBOL-61 Procedure  
DEPRECIATION EXPENSE OR LEASE EXPENSE

- 1000 . IF ASSET-LEASED GO TO 1050.
- 1010 . IF PROPERTY-CLASS IS LESS THAN "A" GO TO ERROR-ROUTINE.
- 1020 . IF PROPERTY-CLASS IS GREATER THAN "J" GO TO ERROR-ROUTINE.
- 1030 . IF ASSET-NEW-WHEN-PURCHASED COMPUTE SUM-OF-DIGITS-EXPENSE;  
GO TO 1070.
- 1040 . COMPUTE STRAIGHT-LINE-DEPRECIATION; GO TO 1070.
- 1050 . IF ASSET-GOVT-COST-FREE WRITE LOCATION-RECORD.
- 1060 . COMPUTE CURRENT-LEASE-AMOUNT.
- 1070 . ADD CURRENT-EXPENSE TO EXPENSE-TO-DATE.

## Chart 7. Sample DETAB-X Procedure

## DEPRECIATION EXPENSE OR LEASE EXPENSE

	Rule 1	Rule 2	Rule 3	Rule 4	ELSE
ASSET-LEASED	Y	Y	N	N	-
ASSET-GOVT-COST-FREE	Y	N	-	-	-
PROPERTY-CLASS IR "A"	-	-	N	N	-
PROPERTY-CLASS GR "J"	-	-	N	N	-
ASSET-NEW-WHEN-PURCHASED	-	-	Y	N	-
WRITE LOCATION-RECORD	X	-	-	-	-
DO	-	Table 5	Table 6	Table 7	-
ADD CURRENT-DATE TO EXPENSE- TO-DATE	-	X	X	X	-
PRINT ERROR	-	-	-	-	X

NOTE: TABLE 5 computes current-lease amount.

TABLE 6 computes sum-of-digits-expense.

TABLE 7 computes straight-line-depreciation-expense.

## Chart 8. Goals for Future Business Languages

1. IMPROVED COMMUNICATION AND DOCUMENTATION
2. INCREASED EFFICIENCY OF COMPUTER PROGRAM
3. REDUCED COMPUTER-PROGRAM CHECKOUT TIME
4. INCREASED ACCURACY IN PROBLEM STATEMENT
5. COMPLETENESS OF PROBLEM STATEMENT



REFERENCES

1. Howard Bromberg, "COBOL and Compatibility," Datamation, February, 1961, pp. 30-34.
2. DOD Document - "COBOL, 1961 Report to CODASYL (Conference on Data System Languages)" (for sale by Superintendent of Documents, U. S. Government Printing Office, Washington 25, D. C.).
3. Burton Grad, "Tabular Form in Decision Logic," Datamation, July, 1961.
4. Orren Y. Evans, "Advanced Analysis Method for Integrated Electronic Data Processing," IBM General Information Manual #F20-8047.
5. Charles A. Phillips, "Current Status of COBOL," Proceedings of the USAF World Wide Data Systems and Statistics Conference, October 26, 1961.
6. Systems Group (CODASYL), "Preliminary Specifications of DETAB-X," August, 1962.

## APPROACHES TO DECISION TABLE PROCESSORS

K. R. Wright

INTRODUCTION

My family and I were out traveling one day, looking at the sights. We came to a scientific museum. Since my sons think they want to be scientists, we stopped to see what we could learn. As we traveled through the exhibits, looking at all the marvels of the modern age and all the fabulous things they could do, we came upon one gigantic piece of equipment. There were wheels, and bells, and arms, and pulleys, and levers; everything working furiously, around and back and forth and up and down. The thing was making a tremendous amount of noise, as though it were accomplishing almost all the work in the world all by itself. It was built so that you could see all the wheels and gears turning and the levers carefully moving back and forth. Everything seemed to be running just fine; then we read the inscription on the base of the big machine. Its specific purpose was just to run. It had no practical use. It was just nice to look at and see all energy being expended.

We didn't want to be in the position of having decision tables nice to look at, a wonderful idea, but not able to accomplish anything. We knew that in order to be useful decision tables needed to be translated into a machine language so that they could be processed by a computer.

TYPES OF PROCESSORS

There appears to be four basic types of processors or methods of converting decision tables to a machine language. These are (1) the manual processor, (2) the interpretive processor, (3) the translator, and (4) the compiler.

Manual Processor

The manual processor is the programmer who sits down with a decision table and translates the decision table into a machine understandable language. By a machine understandable language I mean either a machine code or a language that is acceptable by some other processor.

The manual processor has a number of advantages. Since a person is interpreting the meaning of the entries in the table, the language of the table does not need to be restricted. In fact, as with a standard flow chart, the language of the table can be adjusted to each problem and each individual working on the problem. This requires only the definition of a very few rules to be able to use a decision table. In the beginning this is an enormous advantage

since it means that the theory of the decision table can be tested without having to completely define all the rules and without having to establish a special language.

These advantages, however, tend also to be disadvantages. Since we hope to make the decision table a documentation of the problem, we are defeating one of our purposes. The decision table is a replacement for many flow charts, and it can suffer the same fate as a flow chart. When changes have to be made they can be made in the machine usable language rather than in the decision table. The decision table can end up not being the correct documentation of the problem. If the decision table is not precise, not everyone can understand what it says. It has to be translated by the person who prepared it.

The manual processor does make possible the immediate use of decision tables. A minimum of instruction allows the analyst and the programmer to communicate with each other with a technique that lends itself to precision of definition.

#### Interpretive Processor

An interpretive processor is essentially an object program made up of a series of sub-programs in a machine language. The interpretive program is put into the computer. The decision table, in a machine language, is then read into the computer by the interpretive processor. As the processor examines a decision table it recognizes the various situations that can arise. As it encounters each situation it transfers to the special sub-program that understands this type of situation. This sub-program processes that part of the table, then transfers back to the main program to find out which of the sub-programs is needed to process the next part of the table.

The interpretive processor has a number of advantages. Since each situation in the decision table must be well defined, this type of processor requires a very precise language. But, since each situation requires a sub-program to process it, the processor normally has a very limited vocabulary. Because the processing is done direct from the decision table, the decision table must be kept up to date at all times. Therefore, when the program of the decision tables is debugged, ready to process actual data, the documentation is up to date.

The major disadvantage of the interpretive processor is the operating inefficiency of the program. Since the object program is the same for all problems, it cannot be modified to take advantage of situations as they occur. Therefore, it is normally an inefficient object program and takes more machine time to process a decision table than should be required.

### Translator

A translator is a processor that takes one language and translates it into another language. For example, in DETAB-X the language of the decision tables has been developed so that it is readily translatable into the COBOL language.

This type of processor has a number of advantages. Probably one of the most important is that the writing of the processor is much simpler than with any other type of machine processor. So that the language can be translated, it must be a precise language which is needed to make documentation understandable to others than the author.

Of course, there are disadvantages also. Since the language restrictions used were not developed specifically for decision tables, there are some inefficiencies in the language. For this reason, we have modified the COBOL language slightly to make it more easily usable in decision tables. The placing of an intermediate language--in this case COBOL--between the source language and the machine language gives the programmer a chance to make corrections and modifications to the program in the intermediate language. Thus, there will be a tendency to not keep the decision tables up to date. However, COBOL is a fairly good documentation language so this may not be as much a disadvantage as with other intermediate languages, but should be discouraged.

The insertion of an intermediate language means that the compile time or time from decision table to machine language will be increased. The restriction of going from decision tables to another language will introduce certain minor inefficiencies in the object program. The processors will not be able to make the most efficient object program that could be made for decision table processing.

### Compiler

The last type of processor is the compiler. The compiler takes some kind of source language and translates this into a machine language. This type of compiler has also been described as a generator in that the compiler looks at a statement in the source language and from this generates the instructions necessary for the computer to follow the procedures indicated by the source statement. It is normally referred to as an English language or higher language compiler.

A compiler has many advantages over other types of processors. The source language can be developed so as to be the most effective type language for use with decision tables. The fact that it requires a precise language is in itself an advantage. As the processor would be developed for the specific purpose of processing decision tables, it will be possible to prepare it so that we could obtain the most efficient object program for processing decision tables. If the compiler was written so as to go direct from the decision table

to the machine language, then the corrections would normally be made to the decision tables rather than to some intermediate language. This would mean that the decision tables would be an up-to-date documentation of the problem definition.

The main disadvantage of a compiler is the time required to actually write the processor and debug it so that it is operational. To do an effective job the compiler would take several times as long to get operational as would either a translator or an interpretive type processor. This would decrease our ability to test decision tables in a number of areas at the same time.

### CONCLUSION

Our goal in developing DETAB-X was to develop a language for decision tables that would give us the best possible ability for testing the tables. We could get the biggest range of possible testing if it were possible to use all four different types of processors. On examining the restrictions of the various processors, it appeared to us that the one that was the most restrictive was the translator. This is because a translator requires, for ease of writing, that the language you are translating from be compatible with the language you are translating to. We knew that even though the language developed for the translator might be slightly restrictive to the other types of processors, still they could be fairly effectively used for processing the DETAB-X language. As COBOL is the language that is being most widely implemented in data processing, we decided that the DETAB-X language should be compatible with COBOL. This would give us the widest range of machines and work on which to test the decision tables. This still does not deter the testing of decision tables with other types of processors and with other languages.

There are processors currently available in each of the other categories. At least one program has been written that is interpretive and processes decision tables on an interpretive basis. One translator has been written that translates from decision tables to the FORTRAN language. At least one compiler has been written that compiles decision tables directly into a machine language. A number of installations have done manual processing of decision tables, putting them into such other source languages as FORTRAN, Commercial Translator and symbolic assemblers.

## QUESTION AND ANSWER PERIOD

MORNING OF SEPTEMBER 20, 1962

MODERATOR: L. W. Calkins

PANEL: Burton Grad  
Mary K. Hawes  
Solomon L. Pollack  
Kendall Wright

CALKINS: We have received some written questions, four or five of them, and I will start with these. For others, just raise your hand, I will acknowledge you; state your question; I will then repeat it for the audience and assign it to myself or one of the panel.

The first question is, "Who has written compilers for Decision Tables, specifically?" Ken, I think that falls in your category.

WRIGHT: Well, there have been a number of compilers written. We heard about several of them up at SHARE last week. The RAND Corporation has a compiler. You will hear about that this afternoon. G.E. has a compiler that is interpretative. G.E. also has a compiler that was written from a table into the machine language, as part of their program system for the 225. And our manual processor.

CALKINS: The next question is: "Have you had any experience with decision table processors?"

WRIGHT: As I say, I am a manual processor. I had to write a program for salary distribution. Most salary distribution is quite messy in the logic, so I used the decision tables to describe the logic, and translated from this into the FORTRAN language manually. That was before George Armerding wrote that FORTAB process.

CALKINS: Another question here is, "For what kind of problems are decision tables useful?"

GRAD: It was alluded to about six million times this morning. But basically, it's problems with conditions in it. To elaborate just for a second on it, there must be some sense of alternatives of parallel logic, if you will, of multiple conditions effecting given actions that you will take. It's clear that if there is but a single condition, and it is a "Yes-no" state, there is little to be gained in a technique that is principally aimed at complex decision logic. Because that isn't complex! In general, it is also where you have

an interaction of conditions. Where no one condition of all determines fifty actions, and then some other conditions determine fifty other actions. But where it is joint, it is the interaction of the conditions that determines the actions to be taken. And generally, I should think, also, that the more complex the problem, the bigger the problem, you tend to find greater advantages in decision tables. An analogy has been drawn by some people that the advantages are not linear in a small program that would result in, say, two or three tables or a hundred to two hundred instructions. It just couldn't matter less what you use, almost. When you get to very large programs, it becomes a greater and greater advantage. These are the kinds of claims that are made.

CALKINS: Another question here: "Are processors necessary for the use of decision tables or can you manually code from tables?"

GRAD: Well, Ken has already answered this. Of course, you can manually code from tables. I think, perhaps, the question should be interpreted this way: "Where is the greater advantage in the use of tables?" In the use of them as an automatic input to the machine or as an analysis and documentation tool to provide a problem analysis for the programmer. And I just think it's entirely premature to answer this. It's obvious that people have used tables and have written code from it. One manual processor sits to my right here. Others have, of course. What the advantage is of actually having a processor for it, I don't know. It's one of the things we are going to find out in the next year. Where is the greatest value, and how much relative effect should be put in these different directions?

HAWES: I would like to add a comment. I found that the use of the decision tables, even while you are trying to define the problem in areas that will not be used on computers, has been very promising. I think this is one of the greatest uses, really. Because you have it not only for the computer use, this is only part of the problem; we must not forget the human factor when we are talking about computer systems, and I think this interrelationship is very important. Definitely, even outside the areas that will be computerized, decision tables are very useful.

CALKINS: I have one here: "Are the format restrictions introduced because of DETAB-X, or are they desirable for other reasons?"

GRAD: That, unfortunately, is a very difficult question to answer. Most of the restrictions I described and talked about this morning were developed independently of the DETAB-X specifications. There has been experimentation, as many of you may know, over the past three to five years on this format, and by having tried different kinds of restrictions, or different kinds of freedoms of rules, we find that some lead to errors. It destroys the value of the accuracy area, for example, whereas, other restrictions tend to produce better operational programs and more logical statements. People learn them more readily and use them more readily. In general, therefore, the restrictions were introduced independent of DETAB-X. Nevertheless, certain changes were made

because of the particular language. A simple one in the DETAB-X language as such, is the format introduced there. There is quite a large stub area. This is because you can have quite long statements in COBOL, and since we are compatible with COBOL, you allow for this.

The forms introduced are: a form for limited entry; a form for extended entry; and the size of each of the columns is affected by what can or cannot be done in COBOL and, therefore, in DETAB-X. I mean, they are not the other way around, if I may. I think a restriction for convenient use of tables is to permit you to see things as a whole. If you let the table stretch out too much horizontally or vertically, you begin to lose the ability to see the things any longer. So things introduced into DETAB-X, shortening of words and shortening of some of the operators, are done specifically to take advantage of the tabular format. So I flipped the question, and the restrictions tended to go the other way. They tended to reflect the language rather than the language introducing the format.

WRIGHT: I would like to make one comment. At the SHARE meeting last week, I found out there is at least one installation who has written a processor to go from the data description in COBOL, to COBOL language. They decided that it was easier to do a data description in tabular form and write a processor for the 401 to translate from the tabular form into the COBOL a description statement. This has been written and is operating now in at least one installation.

CALKINS: Are there any questions from the floor?

VOICE: From what I can tell, there is a reduction in words and rearrangement of words in DETAB-X. But the major problems in writing a compiler or a processor, such as COBOL, are still syntactical and semantical problems. How would they differ in this problem? Would they still be confronted with the same problem as the COBOL-type processor?

WRIGHT: The question, as I understand it, was, is DETAB-X going to do away with all syntactical problems when you write a compiler? Is that right?

VOICE: Not do away with all syntactical problems, but how is a DETAB-X processor easier to write than the COBOL processor where the major problems are the semantical and analysis problems? I don't expect it would do away with all of them.

WRIGHT: As near as I can tell, the syntactical problems will be exactly the same. We aren't trying to change at all the syntactics of COBOL. They are all there. The things that we have changed are a few of the verbs. We have added some, and we have made some modifications to make them easier to use in tables. I'm afraid that we are stuck with the syntactical situation with processors as long as we have a formal language.



POLLACK: I will try to cut at your question in another way. If you are going to actually build a compiler directly for DETAB-X, you do not necessarily have to get involved with the syntactical problems of COBOL. It appears to me, for example, you will notice in the data description one of the columns is called "abbreviation," and this is equivalent to the COBOL renames. Now, if you write what Kendall calls a translator, and you want to go from DETAB-X to COBOL-61, you have to get a statement from that table which says so and so, the particular abbreviation, renames, and you then list the other data item. In the case of going directly from the data description of DETAB-X to an object program, you no longer need to do this, and it seems to me that you would have the particular abbreviation right next to the data name for which it is an abbreviation, and you thereby skirt some of the problems of, let's say, the rename clause.

GRAD: Let me give another example. The question is disconcerting for those who still haven't heard the original question. What effect does DETAB-X have upon the compiling of the syntactical problem they have in compiling? Does it have any substantial effect? One of the difficulties, I believe, in preparing compilers for COBOL is the fact that you must completely decompose free form, and many different things can happen next. We have the compound conditions to handle, we have the implied subjects, the implied objects, things like this, particularly in the conditional area. The fact of the matter is, the way DETAB-X is written you are always working on a comparison of two items. They are always going to appear in certain physically known locations. The fact that you are controlling position and location should solve certain problems of compiling, certain problems in the compiler itself. The problem of going from a name, though, to some kind of symbol table is not changed in any way at all. This has no impact. But in some of the procedural statements, particularly in the conditional area, there ought to be some simplification out of eliminating the compound statement, the implied subjects, the implied objects, things of this type.

VOICE: It has been said from the platform that decision tables as a technique are particularly useful for analyzing complex systems problems. On the other hand, it has been stated that as the table extends more and more horizontally and vertically with more rules and conditions, it becomes less useful as a clear presentation of the problem. How are these two statements compatible?

GRAD: I think the answer is clear. It doesn't, that's all. Two different speakers said it, and that's it. The point would be, of course, if you write individual tables larger, you begin to lose perspective. The thing you have to do is to break the problem up into a series of logical segments. If the problem, of its nature, does not sub-divide, you have had it. You have had it, no matter which way you turn. But most problems that have been tackled so far, where there's any attempt to represent human reasoning, human logic, the human thinking process, we find that the designer or manufacturer, the plant supervisor, inventory controlman, is operating with four or five conditional

variables at a time and often no more than fifteen or twenty particular action variables. And so, there is a somewhat of a self-limiting thing that occurs. You can break a problem down, apparently from the experience to date, into reasonable chunks and, therefore, the system does not consist in a complex case of one tremendous table crossing from here over to where Les is, but, rather, it consists of a set of tables; many times ten, twenty, thirty tables, that each express some logical chunk of the problem.

Now, it is also true, when you do that you lose the ability to see the interaction between the tables; you lose some of the values of the decision tables and there's no way around it at the present time that we have been able to discover. Maybe some of you will.

WRIGHT: I think that, maybe, there's an analogy here. It used to be the programmer who said they could not write an entire program of twenty thousand instructions with no breaks. The programmers have learned they can break their problems up into small pieces now; they write small sub-routines and bring them together. If we have to take the same approach with the decision tables, we must break our problem up. The human mind cannot comprehend a complex decision table, anyway.

VOICE: The statement was made that the use of DETAB-X leads to increased program efficiency. I have a two-part question on this. First, increased efficiency over what? Over COBOL? And second, does this efficiency refer to the amount of storage space used or to the execution time?

WRIGHT: The version I was thinking of was that we can write rules for optimizing a series of conditions. These rules have been written mainly in Boolean as a general practice, where these statements are boiled down to the most optimum statements, at least, the number of "ands" and "ors." We can do this. We can write these rules, and we can give them to a processor, and a processor is going to use them all the time. Maybe they do them, and sometimes they don't. Sometimes they use them, and sometimes they don't. So the table gives us the ability to put into a processor rules to optimize the way that the conditions are considered and the ways the answers are used better than a person can do it, because people don't follow a complete or an accurate set of rules; they do it on a random basis. This optimization can be done two ways: We can build compilers now that optimize either time or space; it can be done either or both ways, either optimizing both time and space or getting some kind of agreement between the two based upon our requirements.

POLLACK: I would like to add this comment: You must remember that decision tables have really not been compiled to any great degree and, therefore, not much is known to date. What are the parameters that really make for more efficient programs? With free form statements it's pretty hard to look at them and say, let's go through all of these and see if we can do these more efficiently; see if there are redundancies of certain decisions, and so forth.

Now with decision tables, for the first time you have a potential for being able, number one, to determine if there are redundant decision rules. Two, you now, for the first time, have the ability to state for each decision rule the frequency with which you expect it to occur. In other words, the number of times you expect a transaction to come in to hit that decision rule. So that, if you have a table on which, let's say, there are five decision rules and one of them gets hit by ninety per cent of the transactions and the other four get hit for the remaining ten per cent, on the surface of it, it would seem to appear that one ought to hit that decision rule first. This is only one of the parameters that I have expressed for decision rules. I think, as people think about it, they begin to find others. For instance, in a decision table there may be as many as five to ten conditions, each one of which applies to each decision rule. And there are some in which only one of the conditions is of interest to the decision rule. And in the other case, there may be ten. Now, it may very well be that the combination of the number of conditions which you are actively interested in are, "yes or no," not the "I don't care" type and the frequency with which you expect to hit that decision rule. These are the parameters that will probably determine the order in which you will actually run those decision rules in your object program.

I hope I haven't confused it for you.

VOICE: COBOL seems to figure very largely in the use of decision tables. I have a question that's really directed at the audience. May I ask for a show of hands of those who represent companies that are using COBOL?

CALKINS: The question is to the audience. Can we have a show of hands of all companies represented here that are now using COBOL?

VOICE: Thank you.

WRIGHT: Why don't you ask, how many plan to use COBOL that are not now using it? The second part of that question is, are you really telling the truth?

VOICE: I have a question to ask Mr. Pollack regarding his decision tables; but seeing we do not have the tables with us now, the examples he used, this probably, isn't the appropriate time to ask. Maybe I should leave that for tomorrow. Maybe we'll get a shot at it then.

CALKINS: What is your question, sir?

VOICE: Well, in the one on Inventory, I take exception to the fact that in Rule Three you specified a "No" -- on Condition Two. I claim it should not have been applicable.

POLLACK: That's very good. I'm glad this came up, because this is one of my contentions, that if I had shown that statement to a free form English programmer, he could never have picked it up. I will rest my case.

VOICE: I wondered whether anyone knows whether IBM is working on any processors to translate decision tables into any of its languages?

GRAD: This is very easy to answer. As you all well know, IBM does not discuss any potential future language processors.

VOICE: Some of the original work with the tabular analysis tables was used for the processor, the string of actions that take place after you determine which table to go to. Now, do you say that this should still be done or do you use the strict paragraphs on it?

POLLACK: Those procedures that you have that don't involve decisions should not be put into a decision table. Use straightforward COBOL paragraphs and sections if you are going to be writing your procedure in a mixture of COBOL and decision tables. It's just that simple.

Now, if you have three or four decision rules; again, with whom are you dealing? How good they are at visualizing relationships that only involve about four decision rules. If they can see it, and you feel you don't want to make a table for it, go ahead and write it. It seems to me that anytime you have at least four decision rules you are in a good position to use decision rules.

GRAD: I would like to make a comment. Sol and I disagree practically on everything. We make a practice of it. This is a very interesting point. The question really boils down to: "It's obvious that the decision table format in the case of the unconditional situation buys you nothing new." This is clear. It's nothing. It's no gain for you because it can't. The only possible table, in general, lies in the showing of all terms. Nevertheless, if you were going to feature one format, and you were trying to take advantage of this modular property that's been mentioned here a few times this morning, there could very well continue to be an advantage to you in writing unconditional tables to maintain this modularity, to maintain the continuity of format. It will come no faster for you that way. You obviously aren't going to save any time. It may even cost you a little bit. The question is whether you find it worthwhile for the standardization of the documentation and, for this modular property, to go in this direction.

POLLACK: I am not the only one who disagrees with Bert on that.

GRAD: That's for sure.

HAWES: I know of at least one tabular processor in which their experience is to go to tabular format for everything.

VOICE: Does the DETAB-X concept permit the action of a decision table to modify the current decision table or another table?

GRAD: No. This is a real point. This has been argued all kinds of ways. Now, correct me if I am wrong, my memory tells me that COBOL is nonintrospective, is that correct? Does anybody recall that there is a command in COBOL that will change the COBOL statements? That's not implemented on many of the processors. This is a very important point. This has been argued and discussed. We think that one of the potential advantages, not in just tabular form, but in other languages, is this unalterability. It has the advantage that nothing can be changed in your program, therefore, if you have to overlay memory with other forms, you don't have to bother to write out what you had in memory. We use the phrase "nonintrospective." Does anybody else ever use it? Well, we made it up, like all the other words you heard this morning. We spend hours making up new words. DETAB-X is written as a nonintrospective language.

HAWES: I think we should keep in mind that there are a number of problems in DETAB-X which have not been spelled out even in your manual. This is one of the reasons why this is called Decision Table Experimental. There are many extensions that we are working on which are not yet included. I would say that this also is one of the big problem areas.

VOICE: The reason, it seems to me, is that the approach would tend to make the table balloon up, especially horizontally or vertically, and number two, the fact that this can never be Boolean-optimized is the one glaring weakness, I think, "in the whole approach; that is, the lack of "OR-ing." If you have a series of conditions which are more or less exceptions to the rule in case of A or B or C or D, then do something; all your parameters have to be repeated for every such entry. I think this is one glaring weakness. Whoever heard of a Boolean specification without the "OR-ing" factor.

POLLACK: As you know, a decision rule consists of an "if" condition one and condition two and condition three, etc. Then take actions one, actions two, etc. Now, these various conditions that need to be satisfied are connected by "ands," and as pointed out by the speaker, there are many cases, number one, when you would like to connect your conditions by "or," and even within a decision table, to save a lot of writing, it would be useful to have a series of conditions connected by "or." This is particularly true in those cases where you test, for instance, for error. You want to check that all the digits are numerical, and you go to each of the rules, making sure that all the digits are numerical; otherwise, you have invalid information. Finally, you want a decision rule that says that if the digits are alphabetic or if some other conditions exist, then your data is invalid; this would be really useful.

In my own particular case I have begun to do work on the theory of decision tables, and I would guess that ultimately we will have decision tables that really break up into two parts: one in which everything to the left of some double line are the decision rules that involve "and" connectives between the conditions; and to the right of the doubleline is a decision rule that allows

for the connection of "or's." It seems to me that this kind of extension ought not to be too difficult after awhile, but some further work needs to be done. In the early stages, certainly, we are trying to get people to use decision tables to introduce the idea of both "ands" and "ors."

VOICE: And this is the reason that it tends to be vertical, anyway. The table belongs up. If you had the "or" facilities you could tighten up the table.

CALKINS: I think your criticism is just, and I think the only thing we can say at this point in time is that we are trying to crawl before we walk, that's all. Hopefully, as we learn and devise new techniques, we can improve upon it. This is not a finished product. You have to take it in the light of being experimental; hopefully, some of the people here might find a way of doing this.

VOICE: I don't really think it's a matter of not being developed, I think what you have is a tool for useful analysis. Now this can be developed into a tool of synthesis, that would be the difference, rather than not being developable, because now you state all the conditions and, if you can modify these in some way to arrange for synthesis to take place, you are proving the analysis of whatever the rule is. In our own application, which is an insurance application, we have been using decision tables for a long time. We do not program from them, however. There's a question of documenting certain kinds of management rules, and because of the fact that we did not have to apply it to COBOL we assigned the "or" problems, at least for documentation satisfaction, by having multiple conditions, where a single line will have a condition, "will A OR B OR C," and then, "yes" or "no." We have gotten around this "OR-ing" in this way. But, as I say, this is not program-oriented yet.

GRAD: I think we ought to finish what he was saying if the rest of you did not hear. What they have done is actually put into that row an "OR" statement, "if A OR B OR C OR D" in the stub, let's say, and then in the entry area, they indicate simply "yes" which applies to any one of these things being true. I know of other people who will have introduced the "OR" on that one-line basis, "if A equals three OR four OR six OR eight," and have actually implemented this. It's not so much, I think, a technical problem here, although there are technical considerations. There are questions again of format, physical limits, what you want to allow and how frequently it is used. This is again something you can only feed back and tell us. Do you need the "OR," is it necessary, is it valuable?

As far as this Boolean reduction thing is concerned, however, that has no impact on it, because you could write a processor which would look for result rules that had identical actions and could then stick the "OR" in between those result rules and produce your Boolean reductions.

VOICE: You have to scan your whole table before you can do it though.

GRAD: That's right.

CALKINS: We have time for about one more question.

VOICE: Would anybody care to speculate on how long a period of time before the X comes off the DETAB?

CALKINS: I will stick my neck out. I would say it would reasonably be between six and twelve months. The reason I say that is that, hopefully, we will get some feedback substantiating the position. Assuming that we do get proper feedback and evaluate it, it would be favorable. Then most assuredly we will be in a position to recommend this to the Executive Committee of Codasyl as an addition to the procedure division.

COMMERCIAL AND ENGINEERING  
APPLICATIONS OF DECISION TABLES

H. N. Cantrell

This paper covers our experience with decision tables, from the time we first heard about them, through experiments in different application areas, to our present rather widespread use of tables in systems design and programming. We will discuss some of the difficulties we have had in using decision tables and some of the advantages we think we have gained from them.

A little background in our history and the kinds of computer work we do may be helpful in understanding the scope of our decision table applications. We have used computers in our Department for about ten years. For the last six years we have had a 704 computer. We did our programming in symbolic machine language until FORTRAN became available and then gradually converted to FORTRAN programming. A few years ago we began phasing out of FORTRAN and into a language of our own development, largely because this language includes decision tables and is adapted to both data processing and engineering and scientific programming. Parenthetically, we might remark that the job of writing compilers for a high-level language would scarcely have been feasible for us without extensive use of decision tables in programming the compilers themselves.

In applications, we do a wide variety of scientific, engineering, and business data processing work. Currently our machine load is about half scientific and engineering work and about half business data processing work with a total of about 200 active programs.

We first heard about decision tables a few years ago when we were in the midst of trying to figure out how to program manufacturing planning work. This is a particular type of computer application in that each job consists almost entirely of a large number of decisions and choices. We had already determined that we could not afford to program this work using classical, flow charting methods or simple table-look-up methods. Thus, when we heard of some of the early decision table work which had been done elsewhere in General Electric, we immediately took up tables as the answer to our problems in programming this type of application.

At that time we recognized that decision tables were a new, different, and potentially better way of designing and expressing the logic of computer programs as compared to the flow charting methods we had been using. In making the transition from machine language programming to FORTRAN programming we had saved a lot of detailed coding effort but we were spending as much time flow charting and debugging the logic of FORTRAN programs as we had with machine



language programs. We were quite excited about the potentialities of this new decision table method but we didn't know if it would work for anything other than manufacturing planning applications.

We did know that decision tables were the answer to programming manufacturing planning work, and we had a lot of this work to do, so we proceeded to design and write a decision table compiler called LOGTAB for Logic Tables. We found immediately that decision tables are a very fine way of designing and expressing the logical decisions which must be made in a compiler. Thus our first, real-life application of decision tables was in the LOGTAB compiler itself. Our next applications were in the manufacturing planning work which we had been trying to do.

At about this time we were planning for a large upsurge in business data processing programming. Neither FORTRAN nor machine language coding for the 704 are very well adapted to this kind of work and we did want to be able to use decision tables. We didn't know if tables would work in business data processing programming but we had high hopes. Finally, after a very careful language design effort and very extensive use of decision tables, we invested a man-year of programming and wrote a compiler system for a general-purpose data processing language.

We now had a complete language system, with decision tables for logic, formulas for arithmetic processing, and data descriptions for input and output editing and for tape file handling. Thus with all of our problems happily solved, we ran unknowingly into a major difficulty in decision table application.

To explain this problem, as we finally understood it, we must discuss some of the philosophy of decision tables. The chief value of tables is that they are much easier for people to use than classical flow charting methods. But this assumes that the people making this comparison are equally familiar with both techniques. We were working with programmers and systems designers with years of experience with the flow charting or sequential decision method of designing and expressing logic. These people reported that decision tables were not easier to use and that they could see no advantage in using tables in their work. This was quite a blow since we didn't know for sure that tables could be used effectively in the work these people were doing. Eventually we found, from experience in other areas, that the trouble here was psychological rather than technological. A programmer or systems designer using tables must do his thinking in terms of the parallel relationships between decisions. This is entirely different, and, in fact, incompatible with thinking in terms of the series or sequential relationships between decisions in a flow chart. We were asking people to unlearn a mental process which they had developed over a period of years and learn an entirely new and different thought process. It is not surprising that these people could see no value in using tables. Tables were, to them, a harder way of doing the job.

Having recognized this unlearning and new learning process for what it is, we have attempted to solve it by giving new and relatively hard jobs to experienced people with instructions to do these jobs using decision tables even though this appears to them to be a harder way to work. (We find they are much more likely to recognize the value of tables when doing a hard job.) Usually, after a few months of this work with decision tables, our people have enthusiastically embraced them and continue to do all of their work using this technique.

Up to this time, we had concentrated on decision table applications in the business data processing area. But we now had a language system, including decision tables, which could be used for engineering and scientific work. We were searching for a good, low priority engineering job to use as a guinea pig, when we were hit with an extremely complex, high priority job of writing a computer program whose output would control a three-dimensional contouring automatic machine tool. This was as complicated an engineering-scientific job as we had ever done. We had had experience with similar jobs in the past and were reasonably certain that we could not do this one at all in the time available. This new decision table technique was available but it had never been tried on any kind of engineering-scientific job, let alone one of this complexity. But, since this was our only hope of getting this job done, we decided to put all our effort into a decision table approach for both systems design and programming. The results still surprise us as we look back on them. The engineering, systems design, programming and debugging were all completed in a total elapsed time of four months--at least three or four times as fast as our most optimistic estimates for the job using flow charting methods. Even this is not the whole story. The complete debugging job on this program was done in the last three weeks of this four-month period by an engineer who had never worked on this job before while the engineer who wrote the program was on vacation.

A more complete story of this project is given in the November 1961 ACM Communications in a paper by R. C. Nickerson, the engineer who did the job.

In this job and the previous compiler writing job, we had achieved what we consider to be remarkable performance. Much of the credit for this must go to the use of decision tables, but we did a lot of other things right too. We had very capable, experienced people who were given full authority and responsibility for the job to be done, with a minimum amount of time lost in communications, discussions and approvals. We don't always do this well. Decision tables are very valuable but they aren't a magic wand that makes all of our problems disappear.

Our success with decision tables on these jobs convinced us that we ought to apply them across-the-board on all work, so we immediately included instruction on the use of decision tables in our programmer training courses. We found that our new people had very little trouble learning to use tables.

They didn't have to unlearn past habits and found tables a very natural way of thinking and expressing themselves. Today we have a generation of programmers who have always used decision tables and are turning out work in a wide variety of application areas.

Beyond the points already covered we have reached some other conclusions from our experiences with decision tables.

1. Value.

The value of decision tables, or the advantages of tables over the flow charting, sequential decision method, varies with the complexity of each individual job. Decision tables have no use, and, therefore, no value in a simple, straightforward job which contains no decisions at all. If a job has only a few decisions which are easy to flow chart, then these decisions can be expressed just as easily, and probably more reliably, in tables. As we consider more and more complex jobs the margin in favor of decision tables increases rapidly. For extremely difficult jobs, decision tables may be as much as ten times as effective as flow charting methods. We can't prove this because we can't bring ourselves to do these jobs twice, once with tables and once without, as an experiment, but we don't think 10 to 1 is exaggerated. We have seen complex flow-charted jobs bog down almost indefinitely with logic bugs.

2. Range of Application.

We see no evidence that the application area, mathematics, engineering, finance, manufacturing, compiler writing, etc., has any relation at all to the value of decision tables. If the application has decisions in it, then decision tables are the way to do it. We have found no applications where all of the decisions have to be made, one at a time, in sequence, with actions interspersed between decisions.

Most of our experience with decision tables has been in computer applications but we have also found these tables to be an excellent way to define and express logical procedures of any kind, quite apart from their value in computer applications.

3. Computer Program Design.

One of the striking features of programs written using decision tables is that this technique naturally leads to extreme subroutinization in program design. The program consists of tables, and subroutines whose actions are controlled by the tables. This extreme modularity makes decision table programs unusually easy to change.

Since the final program is going to be made up of tables and sub-routines, it is natural for the programmer to plan and describe the complete decision structure of the program first. The final version of the plan is also the source language statement for the decision tables part of the program, so we naturally obtain a complete, explicit and final plan with all decisions contained in one or more levels of tables. Subroutine programming to implement the actions called for in the tables is now quite straightforward.

In terms of time and effort required, the planning phase often is harder and takes longer than the subroutine programming phase, but the over-all time savings are very impressive.

This natural separation between decision table planning and subroutine programming gives management a series of key events and distinct activities which can be estimated, scheduled, reviewed and measured. This is a big advantage in managing a large systems design and programming organization.

We also take advantage of this separation of decisions and actions in the debugging phase. We find that a trace of table name and column executed provides a very compact but complete description of exactly what the program has done during execution. This has been very helpful in reducing debugging time.

#### 4. Errors in Logic.

People do make mistakes in decision tables, but the error incidence rate in tables is, if anything, less than the error rate in writing formulas or in keypunching. Logic errors are much less frequent than in flow-charted programs and are much more easily detected and corrected.

#### 5. Documentation.

Decision tables are quite understandable by people. Thus, they are an important and explicit part of the documentation of a program. We have been particularly impressed with the ease with which a programmer can take over someone else's job in mid-stream. We have had to do this on several different jobs in various stages of completion. In every case the transition was accomplished with very little loss of time. This has been tremendously valuable to us. In fact, decision tables would be worth using just for this feature alone.

## 6. Object Program Efficiency.

There are many ways of implementing decision tables in actual computer instructions. Some of these are efficient and some are not. The method that we use in our LOGTAB compiler is quite efficient for large tables with many columns and many redundant decisions and actions. It is adequate for most other tables but can be inefficient for small tables in tight loops where the table "overhead" instructions are a significant fraction of the total number of executed instructions in the loop. We believe that it is possible to implement tables with a compiler technique which would give very efficient programs for these small tables and much less efficient programs for large tables. Then with both compiler methods available we could use the best method for each table. We have not yet done this. If we find that a few of the small tables in a program can hurt its efficiency, we express those tables in "IF" statements rather than in table language. This is not much of a problem, so over-all we feel that the use of decision tables has little or no effect on the efficiency of our running programs.

## 7. Learning to Use Decision Tables.

We have tried many approaches to the dual problem of teaching people to use decision tables and convincing people of their value. We find that examples are useful in teaching the format and mechanics of table use and the tricks-of-the-trade, such as looping through a table, using "OR" conditions, etc. But examples simply describe the end product. They do not describe the process of getting to this end product or give much of an indication of the advantages of using this process.

The only good way we have found to learn to use decision tables is to use them. An individual should start with the knowledge of the requirements of a job and design the logic for doing the job in tables. We found very early in the game that a job which already has had its logic described in a flow chart is a poor starting point for a decision table application. It is harder to unwind the flow chart to get back to the basic requirements of the job than it is to express these basic requirements in decision tables.

## CONCLUSIONS

This has been a description of our experience with decision tables over the last two to three years. At the present time we have about 40 computer programs running which use decision tables and about half of our programming staff actively programming with decision tables.

As far as we are concerned, the advantages of the decision table approach have been amply proven and there is no question that this is the way to do systems design and programming work.

APPLICATION OF DECISION TABLES  
TO MANAGEMENT INFORMATION SYSTEMS

Frederick Naramore

SUMMARY

Since 1958 Sutherland Company has been employing decision tables, as part of its Management DATIS System, for documenting management information systems. Major advantages realized through these techniques may be enumerated as follows:

1. The ability to clearly and concisely state system requirements totally independent of procedures and processing media.
2. A uniformly high quality in the statement of system requirements.
3. The ability to associate defined decisions with responsible organizational entities.
4. An effective method for man-to-man communications.
5. The ability to establish an information repository for system specifications.

The composite result may be summarized as the capability for complete and accurate definition of the "what" of a system, independent of, but relatable to, the myriad of procedural details constituting the "how."

DOCUMENTATION CONSIDERATIONS

A prerequisite for any scheme of systems documentation is the resolution of organizational responsibilities considering the interests and technical qualifications of its personnel. This understanding then serves as a basis for establishing document requirements, their particular purposes, and hence level of content. To date there has been an unfortunate tendency to prepare single level systems specifications, with considerable procedural orientation, and use these for both management and technical purposes.

Inherent in any systems development project are three distinct functions. In sequence of occurrence, it is necessary to:

1. Formulate a precise definition of the system's requirements.

2. Design a procedural flow, selecting a particular complement of machines and personnel to operate the system.
3. Prepare detail operating procedures which define the sequential operating steps which process the data through the system.

From the preceding, it may be concluded that the quality of an operating system cannot exceed that of the original definition of the system's requirements. In essence, the procedural system represents an operational plan to satisfy the basic requirements. As an operational plan it is subject to revision based on new mixes of personnel and machines. Such changes, in themselves, do not alter the basic system's requirements.

The availability of bona fide systems specifications stating the "what" of a system as opposed to procedural specifications stating the "how," has been influenced decidedly by organizational structures in respect to separation or consolidation of analysts and programmers.

Generally speaking, an organization which separates systems analysts and programmers recognizes the distinction between the first and third development function. The responsibilities for machine systems design however are not so readily recognized and defined. Consequently, they continue as a source of minor or major irritation by entering into the original systems definitions.

Much of this intrusion by systems analysts in programming procedural areas is the direct result of documentation techniques. Typical documentation, consisting of flow charts and supporting narratives, is a holdover from earlier industrial engineering methods. This combination of material which supposedly represents a definition of systems requirements is deficient in several respects, namely:

1. Detail review and approval by operating management is difficult if not impossible due to the extensive mixing of basic management decisions with procedural considerations.
2. The specifications quite often presuppose procedural solutions prior to resolution of the system's details. Too often such solutions are at the expense of adequate system requirements definition.
3. They are replete with arbitrary sequences inherent in charting techniques, thus artificially imposing constraints on programmers and other procedure writers.
4. The difficulties in indentifying and superimposing changes on the original specification documents presents a task so formidable that it defies effective maintenance.



Organizations which consolidate both systems and programming responsibilities solve the foregoing documentation difficulties through the simple device of not establishing the original requirement for such. In these environments two classes of documents generally evolve.

1. Presentation type material to portray a general definition of the system supported by selected details to imply knowledge.
2. Programming procedures written in the particular language or languages of the assembly or compiling system.

In either environment, it is evident that operating management is not the master of his own house. For all practical purposes he is never quite certain of the degree to which his, and only his, decisions have been incorporated in the object systems.

In addition, the ability to associate management decisions with object procedure statements to facilitate systems changes is virtually impossible without the availability of an individual analyst or programmer who has emerged as the system specialist.

Quite often, the net result is a series of operating procedures which is not readily associated with, or justified by, particular management decisions. Under such circumstances, the problem of change control, including the determination of change impact and assurance of full implementation, is, at the very least, unduly expensive and time-consuming.

The desire to alleviate these types of problems, which stem from inadequate documentation techniques, prompted our experimental use of decision tables early in 1958. The initial objective was to obtain a workable solution to the first level requirement. That is, the communications between operating management and systems personnel, which would promote more accurate definition of system requirements.

#### REQUIREMENTS SPECIFICATIONS

In actual practice the term "systems analyst" has meant all things to all people, hence specific responsibilities vary by job description. One thing, however, is certain. That is, in defining the requirements of a system, an analyst is acting as a licensee of operating management. Acting in this capacity, his first obligation is to positively identify and formally record the policy decisions expressed by operating management. This relationship need not conflict with nor detract from his unique responsibilities in the procedural areas.

Decision tables, or Management Rules as we refer to them, have proved extremely effective in the area of defining basic system requirements. The characteristics of decision tables lend themselves to the logical expression of policy

interpretations independent of procedures. For example, a credit policy requiring knowledge of:

1. Credit rating of customer,
2. Current accounts receivable balance, and
3. Net invoice amount

may be completely described without prescribing, implying, or restricting the procedural steps necessary to execute the policy in the production system.

Management Rules are the formal expression of management dictates stated independently of both processing media and detail procedures. Being independent of ultimate procedures they are independent of each other except in limited situations where sequence is essential to the end result.

Complete requirements specifications, produced by the management systems analyst, are composed of three basic types of information. These are:

1. Element of data definitions,
2. Input and output data descriptions, and
3. Management Rules.

Each will be briefly discussed.

An element of data may be defined as the smallest unit of information which may be separately identified and described. As the basic unit of information, it is the foundation for the rest of the specifications. To assure uniform usage and understanding, defined elements are cataloged in an element library. By including the characteristics and configurations of the element as part of its definition, subsequent system definition is exempted from such details.

Data requirements of the system are grouped in terms of action input data sets, retained data sets and terminal output data sets. As a minimum these are descriptive requirements initially. As particular formats evolve in the procedural phase, they are used to supplement the original descriptions.

Management Rules, in terms of defined elements and data sets, formalize the logic of a management policy by prescribing the particular action or actions to be executed when specific conditions or condition combinations occur.

At first glance these requirements may appear identical to those which are commonly considered as a system specification. Such is not the case. This level

of specification completely excludes processing oriented operators, or verbs, as you may prefer.

Take for example the considerations associated with the refiling of an updated master record. Management personnel have described the conditions or limiting factors under which they will accept or reject various transactions. In addition, they have stated how such activity should alter the permanent records and to what extent selected reports or notices should be prepared. Without explicit procedural statements management has implicitly stated:

1. As a retained data file which has been altered, a need exists to refile the current version.
2. The record can be refiled when no further transactions are present which require this specific record.

The determination of how and when to return the record to file is a procedural matter dependent upon the particular file medium involved in the production operation.

#### DISSEMINATION OF REQUIREMENTS SPECIFICATIONS

Use of decision tables in the preparation of requirements specifications has enabled the development of solutions to two vital problems in the distribution and control of policy decisions, namely:

1. The body of rules can be subjected to manual or machine processes which objectively examine the network of interdependent relationships with the end result being a schematic diagram. It should be noted that these interrelationships are derived from the content of the rules themselves--not from rule connectors.
2. They can be reproduced or converted to machineable records for distribution and filing in accordance with organizational requirements without translation.

The ability to objectively produce a schematic diagram depicting the decision network is of utmost value in several respects.

1. It reveals areas of policy conflict within or between organizations.
2. It affords operating management an opportunity to review their policy decisions for completeness.
3. It provides an impartial roadmap against which the production system can be designed.

Equally significant is the ability to "unitize" the specification components, that is, elements, data sets, and Management Rules; and distribute selectively without retranscription to other forms. Thus, selected duplicates of the original specifications can be furnished to:

1. Operating management in accordance with their respective organizational responsibilities as part of the over-all system.
2. The information repository for change control.
3. The programming and procedures personnel segmented for the particular operation.

The divergent requirements which are satisfied by the original specifications definitely establish decision table structures as an effective multiple purpose tool.

#### APPLICATION EXPERIENCES

Our initial application of decision tables dealt with a highly complex file maintenance operation in 1958. After the expenditure of approximately eight to ten man-months using conventional methods, no accurate specification had been produced. The specifications being produced at the machine run level were ambiguous and contradictory. This was due to the lack of a logical frame of reference to which the analyst could continuously refer. Both the narrative specifications and flow charts were replete with situations where "A" was dependent on "B," "B" dependent on "C," and "C" dependent on "A." Had the equipment been programmed on this basis it could have operated perpetually on any one of a number of transactions.

A crash program to correct the situation using decision tables as the basic form of documentation was completed in approximately three calendar weeks utilizing an average of four analysts. Perhaps one fourth of this time was expended in developing the rationale for completing the tables. By today's standards of disciplined table entries, these were rather elementary and could best be described as a free form mixture of limited and extended entries supported by numerous notes. Two significant points were crystal clear, however:

1. The hierarchy of decisions could be objectively determined and the entries were logically auditable.
2. The use of tables did not imply or impose arbitrary sequences which artificially influenced programming.

Since our original use of decision tables, they have continued to serve as an integral part in documenting system requirements for a wide variety of systems.

Primarily these have been associated with organizations involved in the manufacture or preparation of products for national or international distribution.

Representative segments of the management decision areas that have been reduced to Management Rules are outlined below. These have been selected for illustration purposes not so much for unique problems encountered in documentation but rather for the cross section of management represented. This should serve to dispel oft quoted remarks to the effect, "That is fine for his problems but my problems are different."

The area of accounting spanned general accounting, facility accounting, accounts receivable, billing, accounts payable, cost accounting, standard costs, and product pricing.

Sales areas are represented by such functions as order processing, inventory control, warehousing and distribution, marketing analysis, sales forecasting, and inventory levels.

Manufacturing functions include material requirements (acquisition and control) together with production scheduling of multiple facilities.

Under the broad category of administrative services, the areas of payroll, both wage and hour, tax reporting at all jurisdictional levels, transportation routings, transportation tariffs and import-export tariffs have been completed.

The significance of this list is that many of the related requirements specifications have been produced within the same organization independent of procedural details. In addition, they have been produced in a standard manner to the satisfaction of operating management with diverse backgrounds and interests. Reviews and approvals have been accomplished on the basis of the logic of the policy without introducing the host of backdrop material previously deemed necessary.

An appreciation of the complexity of decisions required might best be realized by considering characteristics of the products themselves. Composite characteristics would include size ranges, quality, seasonal usage, sectional usage, standard and special packaging variances, age and private brands. The influences of these characteristics of course impact heavily on the policy areas of sales, manufacturing and accounting.

#### ADVANTAGES REALIZED IN THE USE OF DECISION TABLES

While the advantages realized through the media of decision tables for the most part have been empirically determined, they are compatible with the experiences of other users.

Undoubtedly the number one advantage in our experience is the ability to effectively record the detail decisions representing operating policies, thus obtaining a problem statement approaching the preciseness of a mathematical formula.

As the processes involved in obtaining such definitions are not confused with procedural development details, actual preparation can just as readily be accomplished by management representatives as by systems analysts or programmers. In practice, superior requirements specifications have been developed by management representatives untrained in procedural details.

Using a limited number of stylized recording standards, the resulting definitions can be manipulated mechanically or manually to produce a schematic of the decision network. Acceptance of the network after comprehensive review of both the schematic and the details of the policy decision constitutes an effective proof of the requirements specifications.

By appropriate cross-referencing or crossfiling elements of data, data sets, and Management Rules an effective repository is available for future change control. For example, the definition of an element of data may be expanded or made more restrictive. Under such circumstances it is necessary to review all rules involving decisions or actions based on the former element definition. Similarly the impact of this change must be reviewed for impact on all data sets of which it is a member.

The schematic of the decision network, in addition to serving as a "proof" of the problem definition, provides an objective framework for design of the necessary production system. Assignment of procedure numbers to the decision rules on the original schematic serves to assure the complete accounting for requirements specification details.

The ability to treat decisions as independent modules results in another major advantage in terms of accuracy. The reproduction of the original tables can be segmented and regrouped to serve as basic specifications for specific procedural areas. The ability to segment and "unitize" this information without translation to another form eliminates the losses in original meaning which are common in translation processes.

Of equal significance is the subject of improved personnel utilization. Its order in the over-all list of advantages does not imply its relative importance. It is merely that the preceding discussion substantiates some of the conclusions.

By following the logical order of a development project, that is:

1. Problem definition,

2. System design, and
3. Procedure preparation,

a reduction of approximately thirty percent in total man-months should be realized. This reduction is attributable to a fifty percent reduction in analysts' time. Table I illustrates the effect on the total effort.

Table I. PERSONNEL REQUIREMENTS

Function	Pers. Equiv.	Using Narratives & Flow Charts		Using Decision Tables		Reduction
		Weighting	Weighted Pers. Equiv.	Weighting	Weighted Pers. Equiv.	
Analysis	1	0.75	0.75	0.375	0.375	50%
Systems Design	2	0.05	0.10	0.05	0.10	-
Procedures	2	0.20	0.40	0.20	0.40	-
TOTALS		1.00	1.25	0.625	0.875	

This reduction in the analysis area is evident in view of the following aspects of problem definition.

Results of a study of approximately 1,000 documents averaging 30 elements per document are shown in Table II.

Table II. 1,000 DOCUMENTS

Use of Elements	Percent
Superfluous	8
Recopied (Document to Document)	73
Decision Purposes	7
Arithmetic Results	12
TOTAL	100

By eliminating procedural considerations from the requirements specification, the management analyst focuses his attention on less than 20% of the entries. This segment in effect represents the pertinent contents of the decision tables in the sense of detail statements of condition-action relationships.

The second major contributor to time reduction is the use of an element library. By establishing standard definitions once, it is not necessary to repetitively and, hopefully, consistently redefine elements of data in each specification. This approach eliminates a major distraction that is present in many specifications; that is, the lengthy details which describe what elements are to be validated and how they are to be validated.

Following the Management Rule approach it is necessary only to indicate disposition actions for invalid information. The test parameters are explicit in the element definition library.

No attempt has been made here to indicate reduction in programming and procedure writing as this has fluctuated widely due to the simultaneous introduction of a host of new programming source languages. To the extent they have been adequately tested and released on schedule, effective reductions have been realized. Unfortunately this has not been the case in our major applications. It is reasonable, however, to project equivalent reductions in the programming areas based on the following considerations.



1. It is not necessary to have programmers participate in the definition phase to obtain background experience which in effect is intended to overcome the deficiencies of normal documentation.
2. Programmers do not have to study extensively an arbitrary processing logic often for the express purpose of substituting their own.
3. Procedure preparation is not subjected to false starts and extensive revisions which occur when problem definition and procedure development progress on an almost parallel basis.
4. The committee approach, that is, joint analyst and programmer development and review of a procedure, has no justification in continuing.

#### CONCLUSIONS

Much of the material presented today substantiates our experience and optimism in the past and future role of decision tables. If our views differ from those of others it is in the sense of the level of usage.

Constantly we hear references to commercial or business languages as solutions to the management-analyst-programmer communications problem. Indeed, remarks have been made to the effect that management can write their own programs thus assuring policy recognition and compliance. While this undoubtedly can be done, such occurrences are unique. In lieu of requesting management to write or at least fully review the ultimate procedures, we must assure, insofar as possible, that that which he has approved resides in the ultimate procedure unaltered.

Our initial effort, as previously stated, was to obtain precise problem definition from management without procedural overtones, this constituting the minimum essential requirement on management. Decision tables have proved effective in this respect. Thus the goals of the initial objectives have been realized.

The second segment of the problem, that is, to assure positively that approved decisions reside unaltered within the procedures, is still to be resolved. As an intermediate solution, the availability of DETAB-X holds more promise than the present business language programs. This is quite reasonable, as a more direct correspondence between specification and procedure can be retained.

A more enduring and certainly more positive solution, however, rests in the ability to accept the policy statements and to automatically produce the procedural system. Undoubtedly many of the conferees and speakers have considered particular approaches. It is our impression that pilot systems of this type are not too far off in the future. Briefly, such systems have to:

1. Develop a decision network relationship (Schematics),
2. Complete data descriptions and test parameters (Elements of Data),  
and
3. Insert the procedural control logic (Table Processors).

Realization of this objective will return operating control back to operating management from whom it has been wrested gradually over the years.

In closing, I should like to offer one last testimonial ascribing to the value of decision tables by relating the following experience. A Sutherland Company representative was recently assigned to develop the requirements specifications of a particular system. Based on an analysis of pertinent procedures and policy memoranda supplemented by personal interviews he committed his newly acquired knowledge to decision tables. Upon completion, the material was presented to various segments of management for review. Undoubtedly this generated the first comprehensive review of a long series of policy determinations. The net result--what had been previously considered and published as nonproprietary information was now endorsed "Company Confidential."

DECISION TABLE EXPERIENCE ON  
A FILE MAINTENANCE SYSTEM

Lynn M. Brown

SUMMARY

A decision table language and computer program pre-compiler were developed at the Insurance Company of North America to facilitate design, implementation and maintenance of a large file maintenance program. The results of this effort indicate that decision tables can have application over the entire systems design-programming area. Decision tables also force a disciplined modularity in the design of a program which can enable a compiler to accomplish some of the program organization function.

INTRODUCTION

The best way to relate experience with decision tables at the Insurance Company of North America is to tell how INA became involved in using them. This is a brief history of the development of a home-grown decision table language called LOBOC, which was concurrent with the implementation of a file maintenance computer program.

About two years ago the Electronics Research Department began implementation of a system to maintain a master policy file for a new type of direct-bill automobile insurance. The keystone of this billing system was a very large and complex program which would run on the computer every other day. The program was to have a life expectancy of several years. During this period, government, management, competition, computer technology and programming-systems errors would force continual revision.

The normal file maintenance program tends to have at least two categories of data error:

1. Detail transactions enter the system which contain incorrect fields.
2. Detail transactions enter the system which are incompatible with the master file in their content.

A system which runs every other day is beginning to have some of the problems that "real time" systems must encounter, and it has a third category of error:

3. Detail transactions enter the system which create a combination of conditions for which the program has no definition of action. These must be detected, recorded and bypassed with no effect on the master file. Analysis of the trouble on one transaction cannot delay the entire system.

The programming-systems personnel assigned to the job were having considerable trouble in laying out the program. The very large number of combinations of conditions possible when a detail transaction was matched against the master file made flow charting difficult. As further definition came to light, the combinations of conditions forced re-design. Original flow charts were redrawn many times.

It appeared that a procedurally oriented language like flow charting was too inflexible for the problem. Since Autocoder and other programming coding languages were also procedural in nature, we were concerned about the economy of revising the machine run over its long life expectancy.

Too many programmers and systems people approach a large problem on the basis that, given complete definition, they can devise a strict procedural flow for the job which will be the most efficient possible for its entire life. This method implies an absence of change both during and after their design work. Over the life of a program written in this manner, the original procedural flow may be patched to the point that a monster is created.

The only thing consistent about business data processing is the need for its systems to be dynamic. Therefore, any strict procedural flow method of design would seem to be in trouble. More flexibility is required.

#### DEVELOPMENT OF A DECISION TABLE LANGUAGE

The main deficiency in the strict procedural flow method seemed to be that it tended to scatter condition testing and action performance. In other words: -test a condition- -do an action- -test another condition- -do another action-, etc. In the complex job INA was attempting they wished to group and analyze condition testing and action performance separately. This might be likened to testing all conditions applicable to a particular transaction first and then picking a specific path through the various actions for that combination of conditions. If the program was unable to complete the condition testing successfully, it meant that the definition was lacking and the transaction record must be earmarked and bypassed. Until the program had determined whether it could handle a particular transaction, no action affecting the master file was taken. By analyzing the condition testing and action performance separately, logic errors could be more apparent.

A program design language was patterned on this concept. It provided for description of a rule which was made up of a statement of conditions and a statement of actions to be taken if the conditions were satisfied.

IF condition (a), and condition (b), and condition (c) are true, THEN  
do action (1), action (2), action (3), and action (4).

Since each of these rules stated an independent combination of conditions, one or more rules could be changed individually with no effect on the rest. A set of rules (a decision table) was assigned to each transaction. The tables could be also changed independently with little effect on each other.

A rule might state, "If the Status Code is normal, and the Billing Date is equal to the Current Date, then prepare a Continuation Notice and go to the Master Control Table."

After writing several rules as English statements, it was found that the programmer kept referring to the same data names, condition names and action descriptions in rule after rule and table after table. Writer's cramp and mistakes began to creep in. Synonym lists to represent these elements as codes were set up.

Example of Synonym Lists:

O1 = The data name "Status Code"  
       "N" used with "Status Code" = The condition name "Normal"  
       "X" used with "Status Code" = The condition name "Exceptional"  
  
 O2 = The data name "Billing Date"  
  
 O3 = The data name "Current Date"  
  
 AAA = The action description "Prepare a Continuation Notice"  
  
 AAB = The action description "Go to the Master Control Table"

Thus a rule could be stated in a "decision algebra:"

O1 = N, O2 = O3, --- AAA, AAB

By developing a format for recording the decision algebra it was possible to put a rule on two punch cards. One card was used for the condition portion of the statement and the other for the action portion. A file of these cards represented definition of the problem to date. It was much easier to change than a flow chart. Machine sorting and collating could aid analysis. Individual cards could be changed at will.

The original intent was to use a listing of this file from which to code the program. However, it soon became apparent that the definition was in a data form that a rather simple pre-compiler could use to generate optimum condition testing. It could produce the result in Autocoder for final compilation on the IBM 7058 Processor. By putting the synonym lists on punched cards, the pre-compiler could write comment description at each place a data name, condition name or action was referred to in the generated coding.

Data description and action performance macro coding were added to the file so that the output of the pre-compiler would contain a complete program input for the 7058 Processor.

As an additional documentative output of the pre-compiler the rules and synonym lists were used to compose English language statements describing the definition of the system.

COBOL compilers received English statements as input and produced coding. INA's compiler received a kind of coding as input and produced English statements, so they called it LOBOC--COBOL spelled backwards.

Theoretically the English statements were good. Technically they were just as readable as COBOL, but they were too complicated to follow. They definitely gave the impression that the sentence and paragraph structure of the English language is not the best medium for expressing complex decision logic. The documentative output of the pre-compiler was changed from English language statements to English language decision tables. This format appeared superior in showing complex decision logic.

#### THE RESULTS

The rather crude home-grown pre-compiler was frozen at this point and used in the implementation of the automobile master policy file program. There were several weaknesses in this version of the compiler and the decision table language. These weaknesses were bypassed by using Autocoder directly in some sections of the program.

1. The linkage control from one table to another was not flexible enough.
2. Autocoder entries to perform actions should have been more disciplined in format.
3. The condition testing was weak in the fact that it was too restrictive in some areas and the generation not optimum from either a speed or space standpoint in other cases.

Measuring the effectiveness of a language on one application is both difficult and unreliable; however, a poll of the experienced programming-systems personnel involved brought forth the following range of estimates.

"The total systems-programming effort was cut about 20%."

"We could not have implemented as advanced a system without it."

There was general agreement that decision tables were valuable not only in the coding, but over the entire systems-programming effort. A twenty percent cut in this area was more valuable than a fifty percent reduction in a coder's efforts. A part of this cut is attributable to a common language for systems-design and programming personnel.

The resulting program was organized in a very consistent pattern by the pre-compiler which made for easy maintenance. The layout of the program within the IBM 705 II core storage and magnetic drum was rearranged automatically by the pre-compiler for each new set of revisions to the decision tables. When sections of the program were allocated to the magnetic drum, all loading and linkage to them was generated automatically. With implementation of this program on a recently delivered IBM 7080, the pre-compiler will lay the entire program in the larger core memory.

Previously, organizing a program within a computer was a design function of the programmer. It is believed that the reason that a rather crude pre-compiler could do this was because of the modular nature of programs written with decision tables. The modularity represents a discipline which is forced on programming-systems designers. A single decision table represents an entity which can be analyzed by a compiler as a unit. Since the machine instructions representing a decision table are a self-contained unit, their sequential location in relation to other portions of the program is unimportant. A compiler can put this fact to valuable use in organizing the entire job, especially if the entire definition is in decision tables.

#### FUTURE APPLICATIONS

INA now has several different types of systems in various stages of implementation using advanced editions of our decision table language and pre-compiler.

For these systems the pre-compiler will have the following new features:

1. Ability to write the entire program in a decision table language.
2. Ability to sectionalize a program to provide for "overlay" of portions of a program.

3. Ability to sectionalize a system definition into one or more programs on a semi-automatic basis.
4. Ability to generate the instructions to perform a particular action, either "in line" or "in one location with automatic linkage," based on a formula using two factors:
  - a. The number of times the action is required by all decision tables.
  - b. The number of instructions required to perform the action.

For example, an action which is only referred to once, or which requires only a few instructions to perform, would always be placed "in line." In any other situation the weights assigned to variables a and b above can be changed to reflect any desired speed and space relationship.

#### CONCLUSIONS

1. Decision tables proved a very valuable tool in the design, implementation and maintenance of a large file maintenance program in both the systems and programming area because:
  - a. The individual rules and/or tables were easy to change with little effect on the remainder of the definition of the program.
  - b. The program design and a major part of the coding were done in the same language.
  - c. The decision table format allowed a compiler to automatically do a portion of the organization of the program within the computer.
2. If decision tables are used exclusively they may enable a compiler to assume some of the program design function.



FORTAB: A DECISION TABLE LANGUAGE FOR  
SCIENTIFIC COMPUTING APPLICATIONS\*

G. W. Armerding

SUMMARY

Scientific computer programs, like business programs, often involve programmed decision logic. Decision tables, which have seen use in business and commercial computer applications, can also be applied to scientific and engineering problems.

FORTAB is a decision table language based on the FORTRAN scientific computing language. Programs written in the combined FORTAB and FORTRAN languages can be compiled by a FORTAB pre-processor program which has been constructed for the IBM 7090 computer.

Initial experiments conducted using the FORTAB language indicate that a decision table language added to a scientific computing language results in a powerful combination of programming tools.

DECISION TABLES FOR SCIENTIFIC PROBLEMS

In describing problems which are solved with the aid of digital computers, we typically classify them into two major groups: "business problems" and "scientific problems."

The classical "business" problem is a data manipulating job. Data is read into the computer; programmed logic determines how the data should be processed; processing is accomplished; the results are then printed. We characterize such problems as being "input-output limited."

The classical "scientific" or engineering type of problem is characterized as being "compute limited." A relatively small amount of data is read; a large amount of straightline or iterative computing takes place, based upon that data; the results are then printed.

In practice, the number of problems which fall neatly into the classical "business" category or classical "scientific" category is small. The usual problem

---

\* The author gratefully acknowledges the assistance of Burton Grad and Thomas Glans of International Business Machines Corporation, who participated in the design of the FORTAB language.

is a hybrid. Many business problems involve relatively long computations within the computer. Iterative routines in business problems are not uncommon. In the scientific realm, many straightline programs do exist, but again a hybrid is usually the case. Some scientific jobs involve great amounts of data and require complicated decision mechanisms to determine what particular computational processes are to be invoked.

The true scientific computer problem is therefore quite different from the "classical" scientific problem. Although many classical scientific problems do exist, the vast majority of scientific and engineering work on computers does involve making programmed decisions. Before and during the processing part of the scientific problem, decision logic must be performed in order to decide what particular computational processes are to apply, what iterative techniques are to be followed, if any, and what actions are to be taken in the case of discrepancies or errors. Even so-called straightline and simple iterative codes contain decision logic which is executed as the computations proceed.

In another type of scientific problem, digital simulation, typical programs consist of complicated logic which determines how the simulation is to progress, depending upon the state of a large number of conditions within the program and within the working data. Simulation programs are largely "decision logic" programs.

Just as the classical scientific problem is rare, so is the classical scientific type of computing installations. In installations where the computing equipment is oriented toward scientific applications, we often find that the same equipment is used for business problems. If the business problems do not warrant computers of their own, or if, for flexibility reasons, it is desired to maintain only a single type of computing machinery in the installation, we find that business-type computing is being done on what we would otherwise classify as scientific-type computing machinery.

In scientific installations, we also often find that the business applications are programmed using scientific-type programming languages. This is done for reasons of compatibility and flexibility. The programming staff and the computers can be flexibly applied to either scientific or business problems as the needs and priorities develop and change.

The above discussion indicates that "business" computing installations do not have a monopoly on programs which contain decision logic. While programmers in business-type installations might feel that decision logic is their forte, we of the scientific-type installations encounter the same type of logic in our programs. We have needs similar to those of persons in the business computing community who are presently concerned about computer programming with the aid of tabular techniques. Decision tables and tabular techniques have a useful place in scientific computing installations. The advantages of such

techniques are: completeness, accuracy and ease of problem statement, reduced programming effort, self-documentation, and readability. All of these apply to scientific problems which involve programmed logical decisions.

#### COMPILED DECISION TABLES

One approach to using tabular techniques in programming is to construct decision tables for the problem at the time of problem formulation. This step is then followed by a manual transcription of the tables into a computer-recognized language which results in the computer program itself. In this type of usage, the decision table supplants the typical flow chart; the programmer works from a table rather than from a flow chart. The table is used not only as a programming aid but also as part of the final documentation of the problem. Thus, as is the case of all documentation, the original tables must be kept current as changes are made to the program itself.

The use of a decision table language imbedded within a programming language has several advantages over the manual use of tables described above. The user of the tables does not need to manually transcribe the tables into computer program logic. This step, with its inherent susceptibility to error, is eliminated by imbedding a table language into a computer-recognizable language. The problem of decision table documentation maintenance is also eliminated. As the programmer makes changes to his program tables, tabular documentation is automatically updated.

#### DECISION TABLE COMPILERS

Several compilers exist which are useful for scientific and engineering computations involving formula evaluation and manipulations of mathematical expressions. Such compilers see extensive use today in almost every scientific computing installation. The primary advantage of these compilers is the ease with which the computer can be directed to perform arithmetic operations, input-output operations, and other procedures typical of scientific computer programs. Where the languages of these compilers are historically weak is in their ability to express complicated program logic in a relatively simple form. Decision tables, of course, provide this ability.

A combination of a scientific computing language with a decision table language would merge the complementary advantages of each. Building a compiler or processor to accept the combined form of these two languages would provide the scientific programmer with a doubly powerful tool: the mathematical language for expression of the computational steps of the problem, together with a decision table language to express the program logic.

This approach has, of course, been applied by the designers of the GECOM compiler language and the LOGTAB processor, both of which were developed by people at the General Electric Company.

### THE FORTAB DECISION TABLE LANGUAGE

Recently, we at the RAND Corporation became interested in the possibility of merging a decision table language with the FORTRAN scientific computing language. Our installation is categorized as a "scientific" installation and we use the FORTRAN language extensively with our IBM 7090 computer. With the assistance of Burton Grad and Thomas Glans, both of the International Business Machines Corporation, we developed a "decision table within FORTRAN" language which we call FORTAB.

In developing the FORTAB language, a number of objectives were upheld:

1. The language of the decision tables should complement the traditional FORTRAN language. The FORTRAN language itself should be unchanged and the FORTAB language should add only those elements necessary to provide a decision table logic facility to the FORTRAN programmer.
2. The decision table language should be easy for the FORTRAN programmer to learn. For this reason, the elements of the FORTAB language should look as much like FORTRAN as possible.
3. The decision table language should be processed automatically for the programmer. He should be allowed to write programs consisting of both decision tables and regular FORTRAN statements. This combination should then be processed automatically, in its entirety, by the FORTRAN compiler and its monitor system.

### AN EXPERIMENTAL FORTAB PRE-PROCESSOR

Throughout its design and implementation, the FORTAB language was considered to be experimental. We wished to test the value of decision tables as applied to scientific problems. Our basic hypothesis was that decision tables would be useful in a scientific computing environment. The experiment was designed to test that hypothesis.

Further, we wished to experiment with methods of adding a decision table language to an existing compiler language. In particular, we wished to experiment with a compiler imbedded in a monitor operating system.

We decided to construct a pre-processor for the FORTAB language. The pre-processor would operate prior to the FORTRAN compiler. In operation, the pre-processor would not process the regular FORTRAN statements, but would merely pass them along to the compiler. Tables within the FORTRAN program, however, would be converted by the pre-processor into FORTRAN statements which would then be presented to the compiler. The compiler itself would require no modification. It would only be necessary to construct a facility for the

translation from decision tables to FORTRAN on an automatic basis, without any overt action on the part of the programmer.

Such a pre-processor was constructed and has been operational in our FORTRAN operating system since July 1962. Because of the manner in which the FORTRAN monitor system handles input to the FORTRAN compiler, it was relatively simple to insert the pre-processor into the monitor. The pre-processor scans each statement of the program; when a table is encountered, it is read, converted to FORTRAN statements, listed together with the generated FORTRAN statements, and then presented to the compiler as FORTRAN statements. No separate pre-processing run is required by the programmer. As far as he is concerned, the FORTAB language has now been merged with the FORTRAN language. Once the programmer has learned the elements of FORTAB, he is free to write programs which are mixtures of FORTAB and FORTRAN. These he compiles and executes just as he formerly compiled and executed "pure" FORTRAN programs.

#### EVALUATION OF FORTAB

In using the FORTAB language combined with FORTRAN, we have experienced the several advantages of decision tables, that is, improved statement of program logic, complete and accurate statement of the problem, reduced programming effort, improved documentation, and ease of usage. We have also experienced several specific advantages of the FORTAB language and its implementation for the IBM 7090 computer.

The ability to present decision tables to the computer through a compiler is a useful feature of our FORTAB pre-processor. Manual use of decision tables is, of course, a helpful tool. However, we have found it beneficial to be able to construct program logic in tabular form and then to present the resulting tables directly to the FORTRAN compiler without manual transcription into either FORTRAN statements or assembly language coding. We believe that the addition of FORTAB to the existing FORTRAN language results in a powerful combination. Presumably, this usefulness would derive from the addition of a decision table language to other compiler languages also.

Training in the use of the FORTAB language is accomplished with little difficulty. FORTRAN programmers can learn the FORTAB language without formal training in a short time. The FORTAB reference manual consists of 16 double-spaced typewritten pages. This reference manual includes a complete program example.

The FORTAB pre-processor lists each table as it appears, in context with the FORTRAN statements which surround it (if any). The listing, which is in an expanded form for readability, thus comprises a major part of the program documentation. This documentation is kept current automatically as the program is recompiled for the purpose of making corrections and changes. Because tables written in the FORTRAN-like notation of FORTAB are very readable, it is also possible for someone other than the original programmer to read the printed listing and quickly understand the logic of the program.

After the FORTAB pre-processor has read a table, it lists the FORTRAN statements which it generates. Although this is not particularly useful information, it is given to satisfy programmers' curiosity. The IBM 7090 FORTRAN compiler supplies, upon request, a listing of the assembly language program which has been compiled. The programmer has complete knowledge of the contents of both the generated FORTRAN statements and their assembly language equivalents.

#### COST OF USING FORTAB

Use of the FORTAB language is not free, of course. From comparative tests we have found that FORTAB programs result in longer compiled programs and thus longer compilation times than the corresponding program written in FORTRAN language alone. A typical FORTAB program takes about twice as long to compile as its corresponding FORTRAN program. The resulting object program occupies about one-third more words in the computer's memory. The running time of the compiled FORTAB program, however, is almost identical with that of the same program written in FORTRAN. This is partially due to the fact that a compiled FORTAB program is more methodical in its flow than are typical hand-written programs. It is not unreasonable to expect that, in many cases, compiled decision tables will run faster than the corresponding program written without the use of a decision table compiler.

Because of the manner of operation of the IBM 7090 FORTRAN monitor in which the FORTAB pre-processor does its work, the actual translation from a table to FORTRAN statements is essentially free. During FORTAB pre-processing, tape input-output is buffered; much of the pre-processor's operations take place during tape movements. FORTRAN programs written without decision tables are not hindered in any way by the FORTAB pre-processor.

The penalties of longer compilation time and larger compiled programs evidenced by FORTAB-produced programs must, of course, be weighed against the advantages of the use of FORTAB for problem solution. Neglecting, for the moment, the reduced programming effort which tabular presentation affords, the longer compilation time must be offset by a reduced number of compilations required for program checkout. Our early experience indicates that the savings effected by the FORTAB language program's reduced number of compilations to checkout will more than offset the increased compilation time. This will be due to the fact that programmers will have a higher incidence of "first-time" and "second-time" correct programs. Programs which normally would require several compilations before logic has been stated properly will now require only a few compilations (probably only one, in many cases). Coupling this advantage with the reduced programming effort required to express a problem solution in the FORTAB language, the net result is a substantial saving of computer and programmer resources.

The extra program memory space required by FORTAB-compiled programs is a fact of life which must be endured. It should be restrictive only infrequently. Our experience has been that, in FORTRAN programs, data requires a far greater proportion of the storage space of the primary memory of the 7090 than does the program itself. Thus a one-third increase in program size will not be noticeable in most cases. In those cases where increased program size would be restrictive, a programmer using tables must reallocate memory to allow the program and its data to fit available space.

## MANUFACTURING APPLICATIONS OF DECISION STRUCTURE TABLES

T. F. Kavanagh

As you have no doubt noted, this is the concluding paper of the day. To a Manufacturing man, the anchor assignment seems particularly home-like and familiar. You see in our normal industrial environment, it turns out that after the researchers have researched, the engineers have engineered, the salesmen have sold, and the accountants have accounted, then it falls to Manufacturing to make the product and prove that it works. Apparently, computer symposiums aren't very different. And, therefore, at this point, you can legitimately ask if decision structure tables have any value to the man in the mill, shop, and factory who must convert these plans and predictions to reality. The answer -- a straightforward yes.

To appreciate the situation, it would be well for us at the outset to share a common understanding of just what Manufacturing is -- as opposed to Marketing, Finance, Engineering, Employee Relations, etc. In general, Manufacturing converts marketing engineering specifications into finished, useable products -- it buys tools and materials, it runs factory machines, it assembles parts, it tests and inspects products, it packs them and ships them to customers. More than just doing the actual work, Manufacturing also concerns itself with developing the most efficient processes and work methods -- and this, of course, is the area of our interest today. Just as there are product design engineers who are interested in the functional soundness and appearance of products; so also there are engineers in Manufacturing who concern themselves with:

- . What is the best machine or process?
- . How fast should it run; what tools should be used?
- . When should we make the parts, how many?
- . How can we be sure that the parts are good?

All of these, and thousands more questions like them, are the everyday province of the engineer in Manufacturing -- in equipment design, methods, wage rate, production and inventory control, quality control, shop operations, etc. Getting better answers to these questions means more efficient shop operations, lower manufacturing costs and improved values for customers. This is the work of Manufacturing.

MANUFACTURING VIS-A-VIS COMPUTERS

There are three factors in Manufacturing's relationship with computers that you should know to completely appreciate the applications stories which I am about



to relate. First, computers are still relatively new to the Manufacturing function. Where they have been employed, the computer has been used in large measure to perform routine clerical operations such as filing, sorting, printing, and the like. Paradoxically, the concept of information processing has not penetrated very far in the world of materials processing. Rarely, for example, does the computer enter into Manufacturing decision-making. The reasons are manifold. The tremendous volume of information and the many complex, detailed interrelationships have made it extremely difficult and costly for Manufacturing people to formalize their logic. Manufacturing still relies heavily on "experience" and "art" as opposed to explicit analytic procedures and quantified design techniques.

Second, computer hardware development is only now beginning to provide the size, capability and cost which Manufacturing needs to install computer equipment at attractive cost reductions -- that is, numerical methods using computers are now only beginning to come up with better, cheaper, faster answers than the "artisan."

Third, today's Manufacturing man knows very little of electronic computers and even less about programming them. Though equally intelligent and bright, many have not had the good fortune to receive the educational background which most of you possess. If computers are to make real inroads, we must find direct, practical ways to show the Manufacturing man what the computer can do for him, and also develop efficient methods whereby he can learn to use them himself. We cannot train enough programmers to program the problems that exist in the Manufacturing function. Even if we had the money, I am fearful that our human resources would fail us.

From this introduction you can gather the fundamental appeal which decision structure tables have for Manufacturing.

1. With decision structure tables, we can quickly teach Manufacturing men now on the job to write their own computer programs, thus avoiding the training of computer programmers.
2. The tabular format of decision structure tables is a reasonably familiar language form. It is not a tremendous departure from the tables which the Manufacturing man has used in methods planning, time standards, lot size determination, sampling and so on. He quickly grasps the power of the structure table to accurately describe logical and mathematical relationships.
3. Structure tables are easy to maintain. Instead of changing all the precalculated answers in all the files, it is often only necessary to change a few tables. In this way the computer is always in position to calculate the up-to-date answers.

4. Decision structure tables provide a simple, uniform format for recording logic which facilitates technical communication within the Manufacturing organization and provides a formal disciplined documentation procedure. This is becoming increasingly important in these days of multi-functional integrated systems. Further, it is a tremendous help in training new people.

#### MANUFACTURING APPLICATIONS

Thus the Manufacturing applications problem is really not one of verb versus verb, or microseconds matching microseconds. At this time we are concerned primarily with making Manufacturing aware of the many practical things computers can do; our problems are demonstrating technical feasibility, proving economic value; defining problems; organizing and maintaining large amounts of data; training people; and so on. Our experience indicates that decision structure tables can really help us in this endeavor. These applications stories on rotors, gears, and inventory control provide some reasons for our belief.

#### CAST ROTORS

General Electric has an understandable interest in electric motors. In one of the earlier Manufacturing structure table applications projects, a study was made of the centrifugal casting process used to make rotors for a line of alternating current motors. As you may recall from high school or college days, electric motors consist of two basic parts: a stationary frame or stator, and a rotating element or rotor. The rotor was made from slotted steel laminations; and copper bars or strips were wedged into the slots. The bars were connected or "shorted" at each end to form a complete electrical circuit. When placed under the energized electromagnetic poles on the stator, torques were set up which made the rotor spin around. The basic theory of electric motors hasn't changed very much; however, if you take apart your washing machine, you will very likely find that the copper bars have been replaced by aluminum. Further, the aluminum was not wedged into the slots, but rather the rotor has been molded together as one solid piece. In addition, odd-shaped protrusions may be sticking out from the end. These fins serve as fan blades for cooling the motor. Many of these cast rotors are made using a centrifugal casting process -- that is, the mold is rotated at high speed so that the liquid aluminum metal will be forced evenly into all the rotor slots, fan blades and other crevices in the mold. In addition, the spinning also helps to prevent the formation of bubbles and voids in the aluminum itself.

In the particular line of cast rotors that was selected for study, over 100 varieties were currently active and, of course, new varieties might appear at any time. The differences in the rotors were basically caused by the different design techniques and configurations used to cool various horsepower motors. To the factory operators this involved different assembly procedures in putting together the molds and also the rotor laminations. In addition, depending on a

number of other variables, there were also differences in the detailed casting procedure.

The first step in the structure table development project was to extract these methods and procedures as well as their supporting logic from widely scattered sources in the current manual system. A considerable portion of the required information existed only in the minds of the folks doing the job. The results were summarized in approximately 60 decision structure tables. These tables covered not only the 100 varieties then active, but also provided the planning logic for some 44,000 rotor configurations then possible.

In addition to describing the factory operating procedure, these structure tables also developed the time standards -- that is, the "allowed" or normally expected operation times. The resulting computer program printed out both the labor vouchers and also the detailed factory operator instructions which told the shop people how to build each rotor. The entire project was completed in six weeks by a man who was then unfamiliar with structure tables, computers or the rotor casting process.

Subsequent to the completion of this work, it was decided to essentially "redo" the project with a new man using flow charts and what were then more conventional programming techniques. This would make about as controlled an experiment as is possible in an industrial environment. One cannot really generalize from one observation, but perhaps you might be interested in a few comparative statistics. The second project took 14 weeks in contrast to six. The second computer program produced similar output, but required a 50% larger object program than was developed using structure tables. However, the structure table program ran one-third slower. The size and speed differences were largely due to different approaches to computer implementation. However, it was clearly demonstrated that both approaches had their merits -- but the 14 weeks versus six weeks really seemed like a good omen.

So much for our experience with cast rotors, now let's turn our attention to another problem.

#### GEARS

Many General Electric departments share a common interest in the production of gears. This component appears in hundreds of the Company's products. However, many are unaware that the common gear is an uncommonly complex thing to produce. Indeed, many engineers -- and some entire companies -- devote all of their interests to the proper manufacture of just this one type of component.

While some simple gears can be molded out of plastic, the more substantial variety -- in which we are interested -- is typically made from flat round metal discs called "blanks." Typically, these blanks are forged individually or cut from lengths of bar stock. In general terms, an average gear might be

number of other variables, there were also differences in the detailed casting procedure.

The first step in the structure table development project was to extract these methods and procedures as well as their supporting logic from widely scattered sources in the current manual system. A considerable portion of the required information existed only in the minds of the folks doing the job. The results were summarized in approximately 60 decision structure tables. These tables covered not only the 100 varieties then active, but also provided the planning logic for some 44,000 rotor configurations then possible.

In addition to describing the factory operating procedure, these structure tables also developed the time standards -- that is, the "allowed" or normally expected operation times. The resulting computer program printed out both the labor vouchers and also the detailed factory operator instructions which told the shop people how to build each rotor. The entire project was completed in six weeks by a man who was then unfamiliar with structure tables, computers or the rotor casting process.

Subsequent to the completion of this work, it was decided to essentially "redo" the project with a new man using flow charts and what were then more conventional programming techniques. This would make about as controlled an experiment as is possible in an industrial environment. One cannot really generalize from one observation, but perhaps you might be interested in a few comparative statistics. The second project took 14 weeks in contrast to six. The second computer program produced similar output, but required a 50% larger object program than was developed using structure tables. However, the structure table program ran one-third slower. The size and speed differences were largely due to different approaches to computer implementation. However, it was clearly demonstrated that both approaches had their merits -- but the 14 weeks versus six weeks really seemed like a good omen.

So much for our experience with cast rotors, now let's turn our attention to another problem.

### GEARS

Many General Electric departments share a common interest in the production of gears. This component appears in hundreds of the Company's products. However, many are unaware that the common gear is an uncommonly complex thing to produce. Indeed, many engineers -- and some entire companies -- devote all of their interests to the proper manufacture of just this one type of component.

While some simple gears can be molded out of plastic, the more substantial variety -- in which we are interested -- is typically made from flat round metal discs called "blanks." Typically, these blanks are forged individually or cut from lengths of bar stock. In general terms, an average gear might be

manufactured as follows: First the blank is rough machined front and back to provide a gripping surface; this also eliminates scale and some excess material. Then the center hole might be drilled, bored and reamed to provide a concentric, perpendicular locating surface for subsequent machining operations. Perhaps a key way or a spline will be formed inside this center hole using a broach. Once these operations are completed the gear may then be finished machined front and back giving the web and flange its final shape. It is only at this point that the gear is ready to start hobbing -- which is one of the conventional processes used to cut teeth. Following this the teeth and other parts of the gear are ground to provide a smooth surface and close dimensional tolerances. In between these operations frequent trips to the annealing furnace are required to relieve the internal stresses which machining has set up within the metal itself. A survey of machined parts shows that the average part goes through five operations; gears average around 30. I think the point is obvious, gear manufacturing can be a complex job to say the least.

In this applications project, the task was to write the decision structure tables to completely describe the operation planning for all factory operations in a large family of complex gears. This is the task of determining which machine shall perform what metal working operations, in what sequence, and with what speeds and feeds, what dimensional tolerances, what tools, and lastly, how long should it take. The results of these decisions were to be furnished to factory operators in the form of printed instructions which contained enough detail to permit them to actually make the gears in question.

At the time that the tables were written, this family of gears was already several hundred strong, however, the objective was to automate the planning for expected additions as well as to simplify file maintenance and clerical operations on the gear planning already in existence. The project to write and debug the 3,000-odd structure tables which resulted took approximately two-and-one-half man years. The resulting computer program contained over 60,000 instructions. The structure tables were written completely by the Manufacturing planning technicians who knew nothing about the world of computers. As a result of their endeavors, the cycle for planning a new addition to this gear family was reduced from four weeks to 20 minutes of GE-225 computer time. The use of the decision structure tables greatly facilitated the documentation of the logic and uncovered many opportunities for standardization. The decision structure table program has now become the official Manufacturing Engineering documentation of the work, functioning in much the same fashion as engineering blueprints. The program is working now and is expected to break even in the first six months of operation.

#### INVENTORY CONTROL

In a completely different type of project, decision structure tables were used to describe inventory control decision roles and also as a simulation language. Here we faced an added difficulty in applying computers. In automatic inventory

control systems, as opposed to mechanized factory planning, we are dealing with more intangible statistical variables over extended periods of time; as a result performance evaluation cannot be as precise or immediate. Really there is no such thing as prototype testing. New inventory control decision rules are installed directly in actual operating systems for a period of time in order to be evaluated. The cost of failure is high. For example, too much customer dissatisfaction or inventory obsolescence can cost a man his job. Needless-to-say, most industrial inventory systems are designed with high safety factors and most innovations are regarded with suspicion. Progress is slow and costly. Further, evaluating new innovations is extremely difficult. If the idea appeared to work the first time it was tried, then it was considered good; if something went wrong, it was considered bad. Often folks are never too certain whether the changes in performance could be positively attributed to the new idea at all. Sometimes other events -- such as a rise or fall in business volume -- were much more directly responsible for the adjudged "success" or "failure" than any influence of the new idea itself.

TRIM -- a computer simulation model to Test Rules for Inventory Management was developed to provide a controlled environment laboratory, where the systems designer can experiment freely with a variety of inventory control decision rules without disturbing the real world. TRIM, like most computer simulation programs, offers three major advantages over real world testing. First, TRIM operates much faster than real time. TRIM can simulate 50 time-periods of inventory systems activity in two to five minutes. Second, because it is a computer model, it is possible to explore extreme situations without risk of destroying the model -- or perhaps more important -- the actual inventory system itself. Third, computer simulation provides a controlled experimental environment where cause and effect relationships can be established with a much higher degree of certainty than can ever be done in the real world.

The best way to describe how structure tables were used in this project is to describe what TRIM is and how it operates. TRIM essentially makes the GE-225 computer behave like a complete single-stage inventory system. It processes customer demands, estimates future requirements, places and receives replenishment orders, purges over-age inventory, cancels over-extended back orders, etc. TRIM also reports how well the inventory decision rules succeed in balancing customer service, ordering costs and inventory carrying charges in accordance with specific weights the user attaches to these measures of performance.

TRIM is controlled by a so-called "master clock." Each significant activity -- forecasting, ordering, etc. -- is assigned a separate alarm clock. The function of this alarm clock is to let TRIM know when this particular activity or transaction will occur next. The alarm clocks are carefully sequenced so that if there should be a tie -- that is, two or more alarm clocks going off at the same time -- TRIM will handle the activities in proper logical, as well as chronological, order. The master clock constantly records "current time" and coordinates all the activity alarm clocks. TRIM's internal activities in their logical order of occurrence include:

1. Purge Obsolete Inventory Subroutine which removes from the on-hand balance any inventory which is over-age, that is, has exceeded its shelf life. The Purge Subroutine examines each entry in the receipt list that has already been received -- i.e., is currently in inventory -- to determine how long it has been in inventory. If this time is greater than the shelf life of the item, this "receipt" is removed from inventory. When all receipts have been examined, the program modifies the on-hand figure and tallies any purged inventory for reporting purposes. This routine occurs first because TRIM would not want to ship any over-age inventory in response to new demands; nor calculate order quantities based on the supposed availability of this over-age inventory.
2. Cancel Obsolete Back Orders Subroutine removes demands which have been backlogged so long, that TRIM must assume that the customer would have cancelled them. When the TRIM Executive calls the Cancel Subroutine, each entry in the back order list is examined to determine how long it has been in the back order state. If this time exceeds the limit specified by the user, then the demand is removed and a cancellation report is printed. The failure of the system to meet this demand is noted by adding the cancellation quantity to a lost units counter. Cancel comes early in the logical sequence of transactions, because TRIM would not want to calculate order quantities based on demand which wasn't really there, and because TRIM would not want to fill these obsolete orders with stock arriving on new receipts.
3. Receive to Stock Subroutine receives replenishment orders and makes the necessary bookkeeping transactions. After a receipt the back order list is examined to see if any back orders can now be shipped. Naturally, in logical sequence, TRIM would want to receive before processing demand.
4. The Forecast Subroutine as might be expected makes estimates of future demand. A forecast can estimate future requirements by predicting the future or projecting the past; TRIM provides for either or both alternatives. Predictions are incorporated through a "base series" which is essentially a list of multipliers. Using the base series it is possible to anticipate seasonal corrections, vacations, changed levels of business activity and other similar influences on future demand. Accurate predictions can significantly improve inventory systems performance.

TRIM also contains a wide variety of built-in forecasting techniques -- moving average, single, double, and triple smoothing -- for projecting past experience. In addition, the user specifies numerous constants and multipliers which further control

forecasting performance. In developing a composite forecast, TRIM first projects past experiences and then modifies this projection with base series predictions. TRIM forecasting also contains some notions of adaptive control. Thus, if forecast errors become excessive, a "panic" forecasting policy can be invoked in an effort to regain control.

5. The Orders Subroutine handles the problem of calculating order points, order quantities, and placing replenishment orders when required. TRIM uses either a fixed order point specified by the user, or a calculated order point. In calculating an order point, TRIM really asks the question: do I have enough on hand to keep demand satisfied until I can get some more -- assuming I pass up this opportunity to order? TRIM poses this question by calculating a proposed order size. If the proposed order size turns out to be zero or less, then TRIM concludes it has enough stock on hand and no order is placed. If the proposed order size exceeds zero, an order is placed. However, the actual size of the replenishment order may be quite different than the proposed order quantity previously developed. TRIM allows the user to impose fixed order quantities, order minimums, order maximums or economic lot sizes.

Since TRIM is a simulator, it must somehow establish when a replenishment order will be received. Lead times can be established three ways:

- (1) Fixed lag time assigned to all replenishment orders.
  - (2) Lag time determined by random selection from a cumulative probability distribution function of lead times provided by the user as initial input.
  - (3) Lag time determined by scheduling a small factory flow shop.
6. The Process Demand Subroutine performs the bookkeeping associated with "shipping" a new demand from inventory. All demand is treated as current demand and shipped immediately if adequate inventory is on-hand. If adequate inventory is not on-hand, TRIM will handle the situation in accordance with partial shipment and back order policies specified by the user.
  7. The Plot Subroutine is one of the optional report features in TRIM which permits the user to get a graphical printout of TRIM's internal operations.



TRIM contains over 100 decision structure tables which generated approximately 8,000 words of programming. It requires a minimum configuration GE-225, card input with on-line printer or punch. It simulates 50 time-periods of inventory system activity in two to five minutes. The original program was done completely in decision structure tables by a two-man team in about three months. The program is operational and has been successfully used by a number of General Electric product departments to analyze existing or proposed inventory systems.

#### CONCLUSION

In summary, we have used decision structure tables in a number of Manufacturing applications. They work well, they appear to offer some definite advantages. But more than anything else it appears that decision structure tables can accelerate the introduction of computers into Manufacturing. In closing, I would like to thank the members of Advanced Manufacturing Engineering, Production Control and Quality Control Service, as well as the various operating departments, who participated in these projects for the privilege of reporting this work. Particular mention should be made of the Company's Computer Department which, as you know, has included decision structure table capabilities in GECOM -- the language for the GE-225.

## QUESTION AND ANSWER PERIOD

AFTERNOON OF SEPTEMBER 20, 1962

MODERATOR: L. W. Calkins

PANEL: George Armerding  
Lynn Brown  
H. N. Cantrell  
T. F. Kavanaugh  
Frederick Naramore

CALKINS: There is one comment that has been threaded through this entire presentation, and I would like to comment on it to you for what it is worth. I will leave you with a question. We have talked about clear and concise documentation. Now, how many of you here are in your second generation of equipment and what kind of left-handed factor did you throw into the estimate of conversion for the lack of documentation?

I do have some written questions that I can start this off with, and we will follow the same procedure that we did this morning. One question that has been handed me is: "Are there any plans for distributing FORTAB to other 7090 users?"

ARMERDING: Yes, we plan to distribute FORTAB through the regular SHARE organization who will distribute FORTAB to 7090 users. We expect that there should be a minimum effort on the part of the receiving people to put this into their monitor system. It replaces one entire section of the FORTRAN monitor. If you have made any changes of your own to that section, you will have to throw your own modifications in, but it shouldn't be too much trouble.

CALKINS: What advantages are obtained using decision tables to prepare test data?

NARAMORE: On our past experiences, aside from the accuracy of the programs themselves (that they have been coded properly), does the system itself represent what the original systems analysts had intended. For some of this problem we have resorted to a very formal procedure for the establishment of test data, and the use of decision tables has been particularly valuable in this respect. That is, the systems analyst can proceed through the logic of each table and at least assure himself that a set of conditions or transaction records, master files, and so forth, is available which does in fact represent the various possibilities of conditions. At the same time, he can then produce predetermined results from these same tables. This has produced, again,

a systems level so that it in turn, being turned over to the programmers, forms the basis for most systems acceptance testing.

CALKINS: Was it not expensive for a user to develop decision table language and precompiler?

BROWN: The program that we were reckoning with at the time (which was the basis for putting together the language) was so large that the development of the language, we feel, over that one program's life will be completely paid for. Both the language and the compiler. It was valuable enough to have it for that one large program, and we feel it paid for itself right there. From that point on it's all gravy, and we learned quite a bit about how to write a compiler in the first place, and we learned where to put more power into our language from the first job. So I think that probably the expense was justified.

CALKINS: Another question here: "Do tables assist in the maintenance of systems?"

NARAMORE: Using the basic decision tables and, as I pointed out earlier, also elements of data and data set descriptions and establishing the repository of systems specifications, again independent of the other procedures, gives a continuing framework against which future changes can be evaluated. For example, any change in value elements and so forth can be examined against the selected rules which treat this element and, therefore, the particular decisions on which the prior definition applied can now be examined to see what impact this has on the present specifications. In that sense, this is considerably easier than poring through numerous flow charts, narratives, or actually going down through the programming details.

CALKINS: Another question here was: "How efficient were the object programs produced by the I.N.A. tabular compiler?"

BROWN: We felt that the coding generated was just about as good as a good programmer would turn out.

Now that depends on your definition of good programmer. It was not quite as good as the next one we turned out. However, this is a small thing, in a way, because a programmer always thinks of what he can turn out as of one point in time if he has all the definitions and if he has everything laid out perfectly. He never reaches that point on a large program. So in over six months we will beat him all hollow no matter how good he is because we have the ability in the compiler to organize the tables in a modular fashion which makes them easier to check and the incidence of error and rerun to correct mistakes is much less. So what he can do as of one point in time is completely theoretical. I don't think he ever does it. So we think we beat him, especially over a program that has a long life expectancy.

CANTRELL: I would like to amplify that a little bit. I think anyone who has ever written a program has had the feeling, or made the statement, that now that I have finished I really know how to do it. I can do a heck of a lot better the next time. This is something that does not seem to happen with decision table programs. I think the reason it happens with non-decision tables programs is that, having completed a job, you now understand it completely and are in a position to plan it completely before starting on it. With decision tables you do plan it completely and, in fact, our programmers that are writing decision programs do not feel that they could do better the second time around.

KAVANAGH: I would like to add another aspect of this which I think is particularly pertinent to those of us who are not really interested in point zero five increase in efficiency and object program.

So frequently you see coding written in such a way that if you stood ten feet away from it and looked on the left hand margin, you would see "I am smart", written instead of the program sheet. If there is a wrinkle, a left-threaded nut or something in that machine that can be possibly used to squeeze a microsecond out of the coding, it has been used, and the devil take anyone who has to come in afterwards and take over that program to maintain it.

I think that those of us who are familiar with, or working with, the structure table area are particularly impressed with the ease with which new people can come in and take over what was done by others. Because it is now logically and completely specified. Where you are faced with high personnel turnover, and in some cases with expanding staffs; in other cases with documenting what "Good Old Joe" has done before he moves on to another and more important assignment, this is extremely important, and I think it's a fact that is often overlooked in the documentation area.

CALKINS: There is another question that I will throw either to you, Harry, or to you, George. Has any work been done to include the decision tables in ALGOL?

ARMERDING: Not to my knowledge.

NARAMORE: Or to mine.

CALKINS: Does anyone here know? I would hate to have one go unanswered. I know of nothing.

CANTRELL: I can make a statement. This is really not an answer to this question, perhaps it's simply a statement that should be made.

You are familiar with the algorithms that are published in the A.C.M., using

ALGOL as a communication language for algorithms, which is another word for theorems, or procedures for a specific purpose. We found sometimes in the mathematical and engineering and scientific programming that we will have algorithms which are a decision table. For example, we have one on a simple method of one variable iteration, Newton's method, which is primarily a procedure. It is not a mathematical statement. The method itself is primarily a procedure. The logical definition of this procedure in a decision table forms an explicit algorithm which is a lot more understandable than the corresponding ALGOL statement.

CALKINS: Are there any questions from the floor?

VOICE: I would like to ask Mr. Armerding about the numerical integration procedure and how he went about using decision tables in it?

ARMERDING: Well, it wasn't my program, it was someone else's.

He had a scheme whereby he would look over sub-intervals of his integration and decide what would be the best method of integrating over that smaller sub-interval of the whole interval which he eventually had to integrate. He would use different procedures, depending on what he found in those sub-intervals. In fact, he used the table in the first place to break up his entire interval into those sub-intervals to decide how the integration was to be done. This is how the program was programmed originally; as soon as he heard about FORTAB he knew it would fit wonderfully in that context, and he redid it; and, indeed, it turned out much simpler.

VOICE: It was a property of the function he could test and make a decision on?

ARMERDING: Right. He could do this dynamically in the subroutine itself.

VOICE: I wonder, if in the manufacturing area anything has been done with decision tables for sequencing jobs in a job shop in order to reduce set-up time.

KAVANAGH: Yes. I think the problem is straight-forward, of course. The problem is, if you go to one task from another, it frequently requires that manufacturing machines be converted. These conversions can be facilitated if the amount of change is minimized. If the same tools possibly can be used, or ones that are very close to it, and if you know the properties of the jobs behind the work station, it is a very simple matter for you to get up a decision structure table to sequence that job in the string or queue behind the work station which has the most desirable properties.

There are a couple of places in our corporation where this has been done in tabular form.

CALKINS: I would like to ask a question in this regard. In the past, the problem has been memory limitation in keeping with the size of job that you are trying to work on. Now, what kind of size have you been able to handle?

KAVANAGH: Well, I think you probably would be able to deduce some order of magnitude from the fact that we are dealing with some three thousand structure tables in the gear project and some sixty thousand in the program, which obviously run out to the size of the GE-225. So, therefore, there is a certain amount of program organization which has to be done.

We have found, however, that we can handle very large programs very well using decision structure tables in the manufacturing area. There is a certain amount of sequential flow associated with the problem, and they tend to lend themselves to segmentation very nicely.

VOICE: I would like to ask Mr. Armerding if the FORTAB could be used on smaller IBM equipment.

ARMERDING: No. It was specifically written for the 7090 computer. There is, really, no reason why you couldn't implement the FORTAB language for some other FORTRAN, however.

VOICE: What is available to the 7090 user in regard to decision tables?

BROWN: I will take a stab at that, since I just came back from the SHARE meeting. Nothing.

CALKINS: As I ventured a comment about documentation, I would also like to venture another thought for you here.

It has often been said and talked about, of how tight my program is and how little memory it uses and how my object time is minimized. People seem to be a little bit afraid of using a little bit more memory or a little more object time. I dare say that if you were to actually and truthfully sum up your costs as they pertain to computer application, probably the cheapest thing that you have is machine time. I am throwing that out to some of the people who like to waive the honor keys as such in terms of programming. Can we have another question?

VOICE: Granted, that the tabular system would serve some of the needs in problem definition documentation. Does it serve also the needs in programming? What about those people who have no TABSOL as G.E. does, or FORTAB? Can the programmer program directly from a tabular format, or must he then draw a flow chart from that?

BROWN: I might take a crack at that, because I think I have two answers to it:

One, if anything, it is easier to program directly from a tabular format than it is from a flow chart. This is for sure.

Two, given the use of tabular formats, I hope we have made the point here that it is not too difficult, to write processors. We wrote one. It took three people; we have never had more than three people on this kind of a thing. All of us here are writing processors to the tune of ten, fifteen, twenty thousand dollars. You compare this to the yearly rental of any computer, and the number of yearly salaries of all the programmers associated with it, it's not very much money.

Two answers: Sure, you can program manually; two, it's not really very hard to write a processor.

BROWN: One other point in this regard. In our beginning we had some system designers lay some things out in tabular format, and then delivered to a programmer. The only kick against the tabular format was (he did the whole job), "What am I supposed to do, sit here and code?" And it was relatively easy to code.

VOICE: Would the panelists comment on the size of the effort required to put into decision tables the program, or problem definition to check design, logical design, of a digital computer such as the 7090 itself. This is, a complex problem with "OR-ing" in it and many, many elements.

CANTRELL: We have given a little thought to this.

First, the problem of designing the logical design of the digital computer is analogous to the problem of defining the logic of a parallel or drawing a flow chart. It is entirely possible, although we have not tried it, that decision tables are a good tool for designing the logic of a digital computer -- the electronic logic.

Second, I would guess that if you could express the internal logic of a machine like the 7090 in a decision table, the very expression of it would be a check of the logic of the machine. You could then, of course, run the thing on the machine and see how it works; but I suspect that the property that decision tables have of making errors in logic relatively obvious to the human being who is making the decision table, would itself uncover most of the logic bugs in the machine.

CALKINS: Are there any other comments from the panel?

KAVANAGH: About two years ago a paper was published by a chap in General Electric about using structure tables in the design of computers. If there are some computer department people here, they might look for the questioner and see if we could get him the full name of the author and the paper number.

experimentation and development. Instead of asking us the question, perhaps we should be asking you.

CALKINS: I can lend substance to that. In Pittsburgh we have installed a UNIVAC 490-Real-Time Computer, operating with a plant twenty-five miles away. It's a research project to see whether or not it is possible to operate in conjunction with recording equipment in the plant, over the leased lines.

VOICE: In our application, we tied to an executive routine a list structure and we were able to take up some of this time that you would ordinarily lose waiting for a device or devices; we used that time to make decisions.

CALKINS: Well, even further, it depends upon the hardware that you are talking about. Let's talk about the ADX, I.T.T.'s message switching equipment, the priority interrupt scheme, where there is a memory location assigned to the line causing the interrupt, and you are at that location in twenty microseconds and the release back to the position prior to interrupt takes, I think, another ten. It's built in the hardware.

VOICE: May I make a comment? In analyzing a real-time system, one of the things that the system analyst has to do first of all is establish feasibility.

Is there enough time to digest the flow of data that's coming into the system. The thing he wants to know, or describe to a programmer, is how to write a program for this thing. "First of all, is it feasible?" Yes, it very definitely is, and I want to show you how it is. I don't want to describe it to the programmer using a flow chart, because this is his function in program design. But if I could use, say, some table like this and functionally describe exactly what things have to be done and in what priority so that you can establish what must be done, and very definitely this can be done in the length of time available with the maximum data flow expected, then you could at least define the bounds of the problem.

CALKINS: Well, I think that you can through the table describe the functions. As to whether you have a true measure of whether you are going to be computer limited, I don't know.

CANTRELL: One comment on real-time programming, at the risk of scaring people off from decision tables. There is one kind of application that we have run into where decision tables apparently do not work. This is the type of application wherein you have decisions to make, but the rate of data coming into the computer is such that you have some data available, which is enough to make some decision and take some action before the next amount of data comes in on which you can take more decisions and take more actions.

Now, a decision table requires that you have all of the data and all of the bases for your decision before you take your action. We have hit a few cases,



CALKINS: Are there any other questions?

VOICE: Does anybody know of any use that's being made now, or any contemplated use, or perhaps any feeling for the difficulties that might be encountered in using these tables for real-time programs in which arbitrary interrupts or arbitrary data rates occur?

BROWN: I have no experience on real-time systems. However, the modular design of a program put together with decision tables seems to be the type of general thing you need available. We have heard mentioned here two or three times that you can put together rather large programs with decision tables. In my understanding of real-time applications, and there are several descriptions of them, it means to me that there is an awful lot of decision-making ability available at one time in the computer at the time any transaction hits it, and it seems to me that this modular approach to the design of what's in there at that time becomes more and more important than it did with what people call batch processing.

CALKINS: May I ask the gentleman who asked that question, even though the application is real-time in the sense that it does take an entry item on demand, are you still not faced with just the handling of the interrupt through an executive routine to get to the location where internal pieces or modules of logic are executed?

VOICE: The essence of the question is whether the mechanics of expressing this in a table -- in other words, now is it obvious when an interrupt can occur? When is it possible to interrupt the table and get back in the flow? If I give this to a management person who wants to describe this program, how can he determine when these interrupts may occur and how they will be automatically processed, and then the flow will continue on some sub-priority process? How can he determine that there are no logical errors in the essence of this process?

CALKINS: Well, I think, really, what must be explained is the function of an executive routine which sets aside the contents of the registers at the end of a given execution on interrupt and the return routine to the point used prior to the interrupt, and then going on. Is this the essence of your question?

VOICE: Perhaps.

CANTRELL: I think you have us here at a point where none of us have done this. Maybe there's somebody in the audience that has, but I haven't heard of them yet. None of us have any experience of putting decision tables to work on real-time problems. We don't know what their advantages or disadvantages are. We don't even know if there are additional language features that are needed. And personally, I think this is a heck of a good field for

like reading cards from the on-line card reader of a 704 wherein the real-time aspect of the hardware is such that you are forced to make decisions, and then actions, and then decisions, and then actions. If you have this kind of a situation -- and we have it very rarely -- I don't know how you can use a decision table for it.

VOICE: As a follow-up to that, wouldn't it be advantageous to have in core a generalized program which would interpret these decision data as data, working against, presumably, a master file and not through this pre-compiling stage, as the FORTRAN's program. This lends itself to real time. We have done this.

We come in with this tabular data; since there's too much to be assembled in core at one time, we have to come in piecemeal in the main master file sequence. It comes in, it's interpreted, and it is executed against the data. Additional data tables are read in, executed against the master file data, and so forth. And it's one pass; rather than having it going through any preprocessing work. It has "OR-ing" and "AND-ing" and everything. If you can do this, isn't it more advantageous than going over the compiling stage with your decision table data?

Can you make a question out of that?

CALKINS: That's the next question. I thought you were going to start that one out with "Four score and seven."

VOICE: We heard this morning about the possibility of having decision tables modify themselves. Perhaps this is just the case where it is required -- for real time. For example, if you make a decision, as Mr. Grad says, based upon preliminary information, this now takes a course of action which now becomes a condition for subsequent information which comes in, and the process is turned over and over again.

CALKINS: Any other questions?

VOICE: I am thinking in terms of the larger problems. Would it not be important to have another section affixed to the table, which would indicate from what table you arrived at a particular table? I think you know what I am referring to. If you have to make a change to a table, then if you have come to that table from other tables, you may have to refer back to see if some other tables will have to be modified, and so forth, on up the line. I was wondering if you have any comments to make along those lines with regard to some of the experiences you had?

BROWN: We do nothing at object time to tell this, but we do have a listing that comes out that tells which tables relate to each other. In effect, it's a listing which indicates the "do's" and "go's". It is sorted by "do's" and

"go's" and the resultant field, where it is going to, so this is helpful. It is not an object time. At object time it is not too difficult to do something like that, because in many tabular systems the linkage between tables is handled by one executive type routine which handles the "do's" and "go's".

So it is possible to put into one location in memory, the identification of the table, or something like that, during the process of hopping from one table to another. This is another advantage in this modularity of going to another place, someplace else, via the same vehicle.

VOICE: But when you are compiling the table initially, you may refer to one table several times. Now, if you have to make a change to this one particular table, it may be that, let's say, five tables had referred to it, maybe there would have to be no changes made to four of them, but a change may be in order for the fifth one. So it seems important you would have to have information handy that would permit you to go back as well as forward. You have a way of stepping down, but you have no way of stepping back up again.

CALKINS: If I might take the liberty of rephrasing your question, I think you have asked this: Having done the application, and a change comes into the logic, if I make that change, how do I know that I have encompassed all of its effects, is that right?

VOICE: Yes.

CALKINS: In other words, when I change this particular point, what kind of chaining effect does it have? In other words, do I know that all logical decisions are correct when I am through with that change?

Do you have any comment on that?

NARAMORE: Our experience with the decision tables as such has not been oriented at a programming level; in other words, a machine-run level. But as part of our procedures, the original sets of decision tables that are produced for a given system are subjected to what we refer to as a leveling technique. This is, essentially, similar to taking a bill of materials for hardware items and developing a listing which references items either upwards or downwards, components to assemblies, or assemblies to components. Taking management rules as such, you could have a schematic, or procedure, which represents each of the independent relationships.

In other words, what tables are dependent on other tables. This is not at a programming level, however; it would be a guide in the sense of changes to know what tables were related.

CANTRELL: I think this is a good suggestion. Probably the only reason it hasn't been implemented is that it isn't necessary to provide this information

in order to compile the program. You only want the program to go frontwards, not backwards; but from the point of view of the people who want to look at it both backwards and forwards, this is very desirable. However, I point out, this same requirement on forward and backward applies to everything in a program; not only decision tables, but formulas and everything else. Probably what we need is a "were used list;" "were used" and "were generated," perhaps. Here is a variable; it is used in all these different places, including this table.

VOICE: Yes. I think all of us who have done any coding have found it very advantageous to have, as Remington Rand calls it, an analyzer. IBM calls it something else.

Where you have to change a particular part of your program, it's very important to know from what other areas in your program this area has been referred to.

BROWN: This was one of our reasons in going into the 7058 processor of IBM.

At the risk of giving a commercial, we get this as an output automatically, and it has a reference in both directions. This probably should be explained. Some other compilers don't have this; they give you an assembly listing, or something like that, but it's a very valuable tool.

VOICE: But you don't have this until you get down to object time, until you compile a program. In other words, you are leaving the burden on the programmer, I believe, rather than on the system generally.

BROWN: The particular format that we write in allows us to sort the cards that the programmer writes up on tab equipment. Since he works directly onto a card format and gets a punch right away, we do have the ability to do it, although we have had very little need to do something like that so far. Usually, we get on to the compiler and actually compile before the program is fifty per cent complete just to get this sort of information out, even though this particular compilation of the program, as far as instructions, will never be used.

VOICE: Sir, I am not sure I fully understand his question. I am just wondering, is this nothing more than getting that close to these decision tables, that these different tables are closed subroutines? I think that's what you are driving at. Because, actually, if you put in your statements "go to" and then come back to "and do this," if you make a change, if you have to perform a certain function, to put a decision table at the beginning of the table, later on you have to do the same thing, you are at a different stage in your program, you have to go back and access this table. Would it not be better to make this a closed subroutine as opposed to repeating this table down below?

CALKINS: Right.

VOICE: And I think this is the question he is driving at.

VOICE: In other words, you leave out the "go-to" statements, or the "do" statements to do that, you would have to build that as an integral part of your program and let the tables stand by themselves?

CANTRELL: You have to be a little careful about making tables closed. Many tables require dozens, or even hundreds, of items of data. And this is a very long key punch.

VOICE: I will buy that. It depends on the length.

ARMERDING: I just want to mention that we do have closed tables in our system, and one of the actions which you can perform on any table is what we call the "perform." That sends you off to another table where you can perform all your conditions and actions for that table. You can do this to any level you like; in fact, these can be referenced by other parts of the program, also, and it will thread its way backward.

VOICE: It's a main part of your program. It has to be.

Let me give you just one illustration.

Subroutines A, B, C, D, and F all require subroutining to perform a certain function in order to complete what is required to be done. You found that G has to be changed to satisfy certain other conditions outside of this subroutine. Well, then it's important to know that A, B, C, D, E, and F have used this particular subroutine and, therefore, you must go back and check to see that the changed G now will satisfy what was originally required of it. Maybe in the case of A, B, C, D, and E this is still the case, but not in the case of F; this is my only point, if that helps to clarify it.

CALKINS: Yes, sir.

VOICE: A couple of speakers have mentioned that the object times of programs using decision tables structure incorporated into another compiler system have been longer than when the compiler system has been alone. I would like to know whether this is caused by the use of the decision table technique, or whether it is perhaps rather due to the forced incorporation of a new procedure into an existing compiler which wasn't designed in the first place to cope with it?

You see, I would normally expect that, if I had an analytical system which gives me a good logical definition from a program, I would get a more efficient object program. That's why I don't understand why these object programs were less efficient than those produced without decision tables.

ARMERDING: In the first place, our programs have not been less efficient. The one where we made the extensive tests actually ran slightly faster in the FORTAB than it did in the FORTRAN program.

KAVANAGH: Your decision structure table processor which was used in the cast rotor did indeed compile down. It was immediate. This was a whole job, this was not married to another pseudo-language. The contrast here, and approach to the computer, was not at language level, it was source. One generated an object program, the other approach was interpretive. Let's put it like this; the interpretive program used considerably less memory, required less programming and did take longer objective time, because you always had to ask what you were doing. So, really, the differences in time here were not due to a shotgun marriage between tables and some existing pseudo-language, but rather in the difference in language approach to the computer itself. One, a pseudo-language which generated an object program, and in another case a somewhat interpretive language, if you would, which has a processor associated with it.

CANTRELL: I would like to summarize these statements.

Given a language which incorporates decision tables in the original design of the compiler, there is no reason why the use of decision tables should in any way hurt the efficiency of object programs or increase the amount of storage required.

There's nothing in the decision table technique, which necessarily has to be slower or use more storage.

We may see lots of examples, because we are throwing these things together rather helter-skelter, where these things are not as good, but this is not the fault of decision tables. In many cases, I think we will find improvements in both storage capacity and object program efficiency through the use of decision tables.

KAVANAGH: Hear, hear.

VOICE: I would like to make an observation -- this is not a question -- in relation to a statement made by Mr. Cantrell that real-time applications present problems because of incompleteness of data at various stages.

A chap working for me who did a certain amount of research on tables arrived at a conclusion that you could set up a priority where rules read left to right, the left rule having a higher priority. In this particular test case, it was possible to go through the table using this, in addition to the technique. This is strictly a suggestion.

CALKINS: Any other questions?

VOICE: You mentioned that decision tables do not lend themselves to modification. In other words, they are not self-modified.

Certainly, if a programmer can indicate a switch in a program, cannot he also indicate a condition which can be tested by a later table?

ARMERDING: Sure. Our programmers do this all the time.

KAVANAGH: This is the essence of a simulation program. We do this all the time. This is the body of it.

CANTRELL: In this self-modification thing, a decision table operates on information, on variables and on constants, too. Now, there is no reason at all why you can't modify the variables that it operates on and have it go down different paths. You may have a decision table which is a loop in which all but one column of the decision table exits back to the table itself. You go through this thing, modifying the variables as you go, until you finally have completed the loop and come out.

I think what these people were talking about in "Introspective Decision Tables," or something like that -- modifying the structure of the decision table at execute time, adding some more columns or some more rows, or changing the type of decision which you make, something which you can't do by changing the variables. At the moment, I am completely at a loss to know what you can't do by changing the variables. You can put dummy variables into counter-columns and blank out other things. We have done a lot of this. We haven't seen the need for introspective decision tables -- if we know what they are.

VOICE: Does the decision table technique do any more than list all possible paths?

KAVANAGH: I will offer one thing. I am sure the others will add more. One thing that it does do, of course, is just not that. It does not list all of the things that could happen by permutating all of the variables; just the things that it will allow. One thing it does for you is to limit the range of possibilities that offer feasible solutions.

CANTRELL: I might cite a for-instance on the possible paths. We had one machine-language written program which had one little hunk of logic in it that wasn't right, and we were trying after the fact to find out what was wrong with it and fix it. After an awful lot of work we finally decided the only way we are going to figure out what this little piece of logic did was to put it in a decision table, so we did. We found it had sixteen possible paths, of which, eight were logically blocked off in the flow chart by things such as the testing of A equals B, and a little further on A does not equal B and, therefore, it wouldn't work. Four more of them were blocked

off by characteristics of the data; that is, A must be greater than ten, but never in this problem can A ever be as big as one.

In the final result we found there were four paths of these sixteen paths that had meaning, and as a result we were able to take the original machine language logic and decrease its complexity by about four to one.

CALKINS: Other questions?

VOICE: Sir, it strikes me that in doing this you were doing one of the things you would preach against by people that don't use this system. The situation might not be as ridiculous as it might appear. In laying out a logical flow chart, you might recognize that here's a condition which we should not tolerate. Maybe it could not exist anyway; but it's not worthwhile going back and saying let me put a stop in here, even though, maybe, I don't need it; so you get some more instructions in there. Now, you are saying that you increased the efficiency of the memory requirements because you eliminated it for the logical table. But wouldn't this really be an investment in something that didn't matter anyway? You are preaching not to worry about using more memory or more time.

CALKINS: This was me?

VOICE: Right -- yet you are pointing out here an example of how you can save memory.

CALKINS: Well, seriously, that's his machine. It's really to each his own. I merely made the comment that, in reality, of the total cost from the time you say, let's mechanize doorknob accounting until you have doorknob accounting running, one of the cheapest things is machine time. Maybe some scientific installations would not agree.

VOICE: I would like to point out that the 7090 costs between four and six hundred dollars an hour, and if in a large installation you save a hundred dollars a day, you have a hundred thousand dollars to play with.

CALKINS: That's why I say the scientific people might object.

VOICE: I can see how decision tables can replace flow charting. I wonder if anybody has any thought whether decision tables will replace PERT.

CALKINS: Will a decision table replace PERT?

ARMERDING: I don't know.

VOICE: I just wonder if he could give me an approximate date as to when this 7090 FORTAB will be available through SHARE.



ARMERDING: We have not found any major bugs in it in quite a long while, so as soon as we can get around to getting it in proper shape to submit to SHARE, we will.

CALKINS: How long? George, do you have any idea how long this will take you?

ARMERDING: Well, we are working on it right now.

CALKINS: Is it reasonable to say three months?

ARMERDING: Yes.

CALKINS: Are there any other questions?

VOICE: Have tables been used for information retrieval? And if so, by whom?

POLLACK: I.A.S. has used it. Advanced Information Systems has attempted to do some work with information retrieval. They are, as a matter of fact, the outfit that was interested in being able to use "ORs" rather than "ANDs" for the kind of thing they are interested in: This OR that OR that, THEN I want this particular document.

CALKINS: Any other questions?

VOICE: Just one comment about this "this and this and this," and this allied subject. We talked about this this morning, the business of operators. It seems to be some marriage between the decision tables and, possibly, parentheses, might give you the operators you want, connectors between the variables involved. There does not seem to be any real apparent way that this marriage could be done right now, but if somebody really wants all the operators, why, you can get them that way.

CALKINS: Well, we would certainly like to get some feedback.

VOICE: There was a question I had this morning, but I couldn't get the floor. This dealt with decisions. The rule -- decision rule, I guess it is. There was an illustration, I think by Mr. Grad.

CALKINS: Is Mr. Grad here?

GRAD: I am here.

CALKINS: All right.

VOICE: It illustrated lines going out the bottom of a table. If this set of conditions were met, then you would take the set of actions below it. Then, he said, you can take one, two or three, or as many as you want, as far as

actions are concerned. This is the part that confused me.

Suppose, now, if this is the case -- I can see, possibly, how you could take action, set number one, action, set number two -- if I understand the illustration properly -- and action, set number three. I think he had that. And then, coming down, you have decision rule two. What set of actions can you take there? Can you take two? And how do you write "I want to take two, and then one?" I was wondering how you would illustrate this in a table?

GRAD: As far as I know, the purpose of the slide in that case was to show that you could not take any action except that set of actions directly below the set of conditions that were met.

VOICE: The one you showed, showed you could take possibly four with one decision rule.

GRAD. Four different rules.

VOICE: One entry point.

GRAD: All that was showing was branching. You came in at the top, depending on which set of conditions were satisfactory. You might go through the first rule or the second or the third or the fourth.

VOICE: Oh. You illustrated it with arrows, that's the part that confused me.

CALKINS: Any other questions?

VOICE: This question pertains to the implementation in the object language, or possible implementation. Does the tabular structure carry over into the object program, or does it decompose into a series of conditional jumps?

CANTRELL: It could be both. In the particular compiler that we have the tabular structure does carry over into the implementation. This is a compiler which compiles bit patterns, one bit per column, and one bit pattern per row. The "AND-ing" and "OR-ing" decisions are then made by logical "AND-ing" or "OR-ing" of these bit patterns together for different rows. So in this particular implementation, the columnar structure and the row structure of the table does carry over into the implementation.

ARMERDING: In our case, as a condition of the table, it is nothing but a series of conditional jumps. But in the action area of the table we perform some logical steps which, to my knowledge, programmers do not use today in the FORTRAN language, even though they could. These steps are easy to set up in the preprocessor. In fact, they work quite nicely to test whether a particular action is to be taken or not; so we use them at each point in the actionery table.

CALKINS: Well, it's a little after five o'clock now. Before we break, I just want to emphasize again for those of you that will not be with us tomorrow that the focal point of contact that has been set up for your comments or your work or your criticism is Mr. Sol Pollock of the Rand Corporation, 1700 Main Street, Santa Monica, California.

I hope that you have enjoyed this day as much as we have enjoyed putting on the program for you. Don't sell this thing short. Don't take a quick look at it and say: Well, I am just going to by-pass it. Give it a try. Because we do need your help, and I think that there are some worthwhile things here. So, please, give it a try.

Thank you very much for your attendance.

Meeting adjourned.