Language ~~[illegible]~~ Structure

A PROPOSED PROGRAM FOR RESEARCH ON TWO-DIMENSIONAL PROGRAMMING CONCEPTS

Burton Grad
Programming Systems
International Business Machines Corp.

March 1, 1960

TABLE OF CONTENTS

References

(1) D.D. McCracken, et al; Programming Business Computers; J. Wiley, 1959

(2) B. Grad and R.G. Canning; Information Process Analysis; Journal of Industrial Engineering; November-December, 1959

(3) Kemeny, Snell, Thompson; Introduction to Finite Mathematics; J. Wiley, 1958

(4) Harold Wolpe; An Algorithm for Analyzing Logical Statements to Produce a Truth Function Table; ACM Communications, March 1958

(5) Orren Y. Evans; Advanced Analysis Method for Integrated Electronic Data Processing (Draft); not published, 1959

(6) R. W. Murphy; A Definition of Block Diagrams; IBM Report IR-00065, 1956

(7) J. Jeenel; A Standardized Representation for Business Problems; Watson Research Laboratory Report, 1958

Section A - Two-Dimensional Programming

It is the purpose of this paper to discuss the concept of two-dimensional programming. This implies some non-serial programming structure to permit taking advantage of the ability of people to see relationships in two-dimensional form. While it is true that a sequence of statements can describe uniquely any operational procedure, this is really not the most important criterion. The two critical elements are:

(1) Does the representation technique provide for ease in preparation and communication? Is it a "natural" form for humans to prepare?

(2) Does the representation technique provide advantages in terms of preparing appropriate Processors? In other words, will the Processors be simple or faster or will the Processor running time be lessened or will the resultant object program be faster in operation or require less memory space.

Examination of tools used to date by systems designers, procedures analysts and computer programmers gives a revealing insight into the desired structure of a representational scheme which is properly "human-engineered". There are four popular forms currently in use for systems description:

(1) Schematic Flow Charts: These illustrate, in essentially a two-dimensional form, the significant systems elements, using lines and connectors to show interrelationships among these elements. The concept of precedence is established either through converting the lines to arrows or by conventions such as flow from left to right and top to bottom. This form has been extensively used by computer programmers and factory layout personnel. Often special symbols are adopted to represent a particular class of operation and typically extensive abbreviation is required to fit the procedural description within the available symbol forms. A sample schematic flow chart is shown on page B-5. A good explanation of this type of charting is given in chapter 3 of reference (1).

(2) Serial Flow Charts: This technique permits only one direction of flow,often from top to bottom. Again, various symbols are used to characterize the different types of operations and special coding is introduced to handle this reference to branch procedures. There are a number of minor variations on this basic theme, but all share the common concept of restrained arrangement of symbols. Serial flow charts are used for process descriptions and paperwork procedures diagrams. One such system is described in detail in reference (2).

(3) Logical Equations: Used primarily by design engineers for complex electronic equipment, the application of Boolean Algebra has grown considerably since 1940. Tending to be highly symbolic and abstract, this format permits various sophisticated techniques to be applied leading to systems minimization. Unfortunately, this approach (together with the extensive use of algebraic formulas) apparently leaves most non-technical personnel somewhat dubious and does not provide a suitable means for communication and reference. Conversely, the essential simplicity and analytic structure of logical equations do much to recommend it.

(4) Tabular Arrangements: In some areas, a tabular form has been adopted in order to clearly show the relationships between sets of conditions and sets of actions or results.

Since this paper is centered around the concepts of two-dimensional programming as embodied in tabular arrangements, we will explore a number of examples of this type of approach.

The foundation for much of the current work can be traced to the logical truth table as described in reference (3). Though used as an analysis tool (rather than directly for programming), this format has offered systems designers a technique for avoiding ambiguity and insuring comprehensiveness. Used in conjunction with logical equations, it provides a clear, easy-to-understand framework for describing and communicating analysis material. In general, a truth table consists of a series of columns in which the independent variables are used as column titles and the various combinations of Truth (T) and Falseness (F) of these variables are itemized in these "condition" columns (see Figure 1).

Figure 1

$$(a \vee b) \wedge (\overline{a \wedge b}) \rightarrow c$$

a	b	avb	a∧b	$\overline{a \wedge b}$	c
t	t	t	t	f	f
t	f	t	f	t	t
f	t	t	f	t	t
f	f	f	f	t	f

These columns are separated from a series of intermediate columns by a double line. These intermediate columns are titled by the particular portion of the initial equation whose truth or falseness is being analyzed. This is always done from the simplest to the most complex relation. Finally, the result is again separated by a double line and marked true or false as appropriate.

There is a sense of totality and straight forwardness in this format which is appealing to many systems analysts. For example, Harold Wolpe of IBM (reference 4) used a form a truth table to explain the operation of a relatively elaborate algorithm which he devised for automatically handling logical equations.

A direct outgrowth of this concept is described in reference (5) by Orren Evans of Hunt Foods and Industries. As part of his excellent paper describing a comprehensive set of techniques for systems analysis (systems flow charts, data layout, field definition, etc.) he uses a "Data Rule" concept. This is covered by an example shown in figure (2), below:

Figure 2

Rule No.	Prior Rule No.	Freq.	C_1	C_2	C_3	A_1	A_2	A_3
001		100	Y	Y		Y	Y	
002		30	Y	N	Y	Y		Y
003		5	Y	N	N	Y		
004		2	N				Y	

C_1, C_2 and C_3 each represent some conditional statement such as: Due~Balance + Amount~of~this~order $\leq$ Credit~Maximum. In each column a Y (for yes), and N (for no) or a blank(for "does not matter") is shown. To the right of a double line a series of Action columns are used. A_1, A_2, and A_3 each indicate some particular action like:

Mark Order "OK to ship"

A Y is used to indicate that this action is to be executed while a blank indicates that it is not to be carried out. Each row is called a Data Rule and has certain identifying material to the left of the condition columns. These are: rule number, which is the row number; prior rule number to indicate precedence relationships; and frequency, which denotes the number of times per week (or month or year) this particular data rule will be satisfied. The structure is such that one and only one rule can be satisfied for a given set of input values and the sequence of analyzing the Data Rules is not important in determining the proper Data Rule. This work has been presented to the Intermediate Range Task Force of CODASYL (Committee on Data Systems Language) and is presently being studied by this group.

While this form has much to recommend it from an analysis standpoint, there are a number of questions which can be raised concerning its usefulness as a programming device:

(1) Since each condition statement must result in either a "Yes" or a "No" answer, extra columns are needed to handle "Or" values and multiple ranges. For example, C_1 might represent: Marital- status is "single"; C_2: Marital- status is "divorced"; C_3: Marital- status is "married". Suppose the logic is as follows: If Marital- status is either "single" or "divorced", then put 1 in column 17; if Marital- status is "married", then put 2 in column 17. Figure (3) represents the Data Rule table needed for this problem.

C_1	C_2	C_3	A_1	A_2
Y			Y	
	Y		Y	
		Y		Y

It is apparent that this could become a serious problem as extensive multiple ranges entered the picture. It is also evident that slightly varying alternative actions can cause the same difficulty. This also may result in a more than linear increase in the number of rows required, since provision has to be made for all logical combinations of the conditions. On this basis, I believe that there is a major weakness in the handling of condition and action statements.

(2) The tables tend to be quite empty and extremely space consuming. In his write-up, Mr. Evans suggests one physical solution to this problem through multiple column identification. However, I don't believe this comes to grips with the underlying problem.

(3) The existence of a third state (blank for "not significant") prevents the direct use of a binary representation for the individual Data Rules. This binary coding would obviously offer very attractive memory reductions together with the possibility of direct binary word manipulation to detect the appropriate solution row. Further work in this direction might prove valuable.

(4) The table-to-table flow is not explicitly defined, therby leaving at least one critical aspect of a total data system open to question.

(5) Apparently, Commercial Translator Statements could be used as the language of the condition and action statements though this then requires the power of a Commercial Translator Processor to provide an object program.

(6) The connective between Condition Statements is only "AND" and the only sequence for executing action statements is that implied by the order of their listing.

In spite of these drawbacks, this technique does seem to offer many of the "human-engineering" advantages which we seek in a two-dimensional programming system:

(1) There is implicit indication of the path to be followed on successful _or_ unsuccessful completion of a test. On success you continue across the current row. On failure you drop to the first test in the next row.

(2) There is a built-in error detecting function. If no solution is found, then failure on the last row could kick the program into a special error reporting routine.

(3) The truth table features aid in preventing and detecting logical errors or omissions.

(4) The formal structure is an aid to program communication.

(5) Through proper sequencing of the columns and Data Rules, a reasonably efficient operating procedure could be evolved.

There is other work in this direction which may be of use to us. For example, Bob Murphy of IBM proposed in 1956 a similar tabular technique for stating logical decisions without the restraint of explicitly defining all procedural sequences (as has to be done in a flow chart). His proposed technique had the same general properties as the Evans' work described above except that he used 0 for N and 1 for Y. He also experimented with a construction which permitted multiple success rows. The concepts which underly this work are described in reference (6). In a different direction, he explored briefly the use of a single column to represent multiple states or ranges of a particular variable. This is shown in figure 4. It appears that this might solve one of the serious problems in the Evans' approach.

<u>Figure 4</u>

Marital Status		A_1	A_2
Single		X	
Divorced		X	
Married			X

In 1958, Joe Jeenel, also of IBM, proposed a system delineation technique which included a modified truth table for logical decision rule description. He also presented a tabular approach to the control of program segments and loop hierarchies. This concept is explained in reference (7).

In 1957, Perry Crawford of IBM led an extensive study involving a full description of the various procedures involved in a particular customer application. In certain parts of the system, the rules were so complex that a tabular definition of the logic was used. One of the charts is shown in figure 5 (next page).

This is a far more compact representation of the problem than could have been obtained through the Evans' technique. However, it still has numerous weaknesses in terms of ease in preparation, ease in understanding, and efficiency in processing and operating.

Another area of tabular development has been in the field of product standardization. There is a well-known form used called a Collation Chart. This is nothing more than a listing of values for various critical specifications in the top rows of the sheet (see figure 6 below) and the names of the parts down the left-hand column. In the various intersections, the appropriate drawing number is entered. A dot is used as a horizontal ditto mark. Oftentimes the quantity, if variable, will be shown within the intersection. Otherwise, it appears adjacent to the part name.

Figure 6

Collation Chart for Electric Clock

Voltage	110	110	220	220	220	220
No.of Hrs.	12	12	12	24	24	24
Radium Dial	No	Yes	No	Yes	No	Yes
Glass	37B40	.	.	.	.	.
Case	37B50	.	.	.	.	.
Face	37B60	37B61	37B60	37B61	37B62	37B63
Hands	37B70	37B71	37B70	37B71	37B70	37B71
Gears	37B80	.	.	.	37B81	.
Motor	37B90	.	37B91	.	.	.

SHIPPING SCHEDULE DETERMINATION

CONDITIONS:

Stockage	Delivery	Availability	Further Conditions	Action	Shipping Schedule Date
S	DI or DN	QA		Ship at once	Today
		QN		Back order	DRO
	DD	Not applicable	OH-QRP $\geq$ QO	Defer order without reserving	DD
		Not applicable	OH-QRP $<$ QO and DD $\geq$ DRO	Defer order without reserving	DD
		Not applicable	OH-QRP $<$ QO and DD $<$ DRO	Defer order without reserving	DRO
NS	DI or DN	QA	QA $\geq$ 1/4 QO	Ship at once	Today
			QA $<$ 1/4 QO	Defer order and reserve	Today + SLT
		QN		Suspend order and order replenishment	Today + SLT
	DD	QA	QA $\geq$ QO	Defer order and reserve	DD
			DD $\geq$ Today + SLT and QA $<$ QO	Defer order and reserve	DD
			DD $<$ Today + SLT and QA $<$ QO	Defer order and reserve	Today + SLT
		QN	DD $\geq$ Today + SLT	Defer order	DD
			DD $<$ Today + SLT	Defer order	Today + SLT

Figure 5

A similar approach has been used to simplify and standardize shop routines and time standards.

All of these tabular techniques offer a natural mode for a two-dimensional programming language. It seems apparent, though, that the use of the intersection blocks for more than just a true-false indicator would extend the span of the table and might provide significant memory reductions. Since there is a variety of particular problems within the framework of a computer program, it may be desirable to analyze tabular formats for each of the key modes of operation: Input, Output, Formula Evaluation, Decision-making, Search, File Maintenance, and Supervisory (Executive).

Given this background of material, the balance of the paper will be concerned with particular aspects of the problem of creating a suitable two-dimensional programming language:

(1) Section B discusses various address modes. It is evident that if a "fixed" format is to be used (like a table) then a standardized address (or operand) system will probably be required. The conclusion of this section is that a two-address logic seems to be a reasonable solution to a two-dimensional programming system.

(2) Section C is brief analysis of the concept of controlled two-directional branching and its impact on the instructions needed in a two-address system.

(3) Section D is concerned with relative addressing and "contained" constants. These techniques make a programming language easily separable so as to permit a segmental approach to debugging.

(4) Section E describes a suggested minimum language embodying the principles described in the previous sections and then briefly indicates a few of the more important extensions and sophistications possible.

(5) Finally, Section F recommends a study program aimed at developing a useful two-dimensional programming language.

Section B - TWO ADDRESS LOGIC

In considering a two-dimenstional programming scheme, the number of operand addresses can be of significance. Most computers have been constructed with a one-address logic. Examples include all of the IBM 700 Series, Univac I and II and Burroughs 205. The IBM 650 is a special case where a second address is included in the instruction word just for instruction sequence control. Various of the newer computers have used a multiple operand address logic. The IBM 305, 1401, and 1620 as well as the NCR 304, Honeywell 800 and Univac 1103 all use two or more operand addresses - sometimes the number of addresses is variable.

It is the purpose of this Section to explore one, two and three-address logic for various classes of instructions and to try to show certain of the advantages and disadvantages of each mode of operation.

Any instruction system must explicitly state in each instruction the operation to be executed and must also state, either explicitly or implicitly, the memory location of the field (s) to be operated upon. In a one-address machine the basic instruction format is:

```
      x...x               x...x
Instruction Code      Field Address
```

In operation, the computer's control element recognizes the instruction code and executes it using the information stored at the field address indicated. The computer then proceeds to the subsequent instruction location (which may not be in numerical order). There are many variations on this theme with index registers, partial word definitions, additional control data, etc., but essentially all single address machines have this basic pattern.

For three-address machines, the basic construction is:

```
   x...x          x...x          x...x            x...x
Instruction   Field Address  Field Address    Field Address
   Code            A              B                C
```

The general mode of operation is for the computer to carry out the operation indicated by the instruction code on the information stored at locations A and B and then either store the results at location C or switch control to location C. The same comments relative to variations is also applicable here with the added complexities of indexing multiple fields, defining partial word lengths for multiple fields, etc.

A two-address machine would have its instruction word composed as follows:

x...x	x...x	x...x
Instruction Code	Field Address A	Field Address B

Operation would vary considerably depending upon the nature of the instruction code. To illustrate, various classes of two-address instructions are noted below; one suggested operation and mode for representative members of each class is then described:

1. Move instructions -
   any instruction which moves information from one location to another. Examples include:

READ	Move information from designated input source (A) to destination location (B).
WRITE	Move information from source location (A) to designated output unit (B).
ASSIGN	Move information from source location (A) to destination location (B).

   These instructions could, of course, specify the movement of partial words or multiple words at once.

2. Relational Instructions -
   any instruction which compares two fields of information.

   Compare Greater Test to see if the contents of Field A are greater than the contents of Field B.
   Many other relational comparisons are possible, smaller, equal, not equal, greater or equal, and smaller or equal. Their operation mode would be identical to the Compare Greater instruction. Another possibility would be to have a generalized Compare instruction which would set a series of binary indicators like greater, smaller, etc.

3. Branch instructions -
   any instruction which changes the normal sequence of instruction execution.

   Branch Based on the contents of location A either switch control to location B or continue the normal operational sequence. Location A might designate some type of memory which has been preset by a previous test such as equals, not equals, overflow, etc.

4. Arithmetic instructions -
any instruction which performs an arithmetic function like add, subtract, multiply, divide, exponentiation, sine, cosine, etc.

Binary operations:

ADD Add the contents of location A to the contents of location B. Store the result in location B. The same approach would be followed for subtract, multiply, divide or exponentiation.

Unary operations:

SINE Determine the sine of the contents of location A. Store the result in location B.

Certain normally binary operations can be restated as unary operations if it is useful because of frequency of application.
For example: $(A)^{(B)}$ can be restated as:
Square Root (A) if (B) = 1/2

Other examples of unary operations are those which are performed for decimal (or binary) point location or for format modification (either input or output).

Shift Right (I) Shift the information in location A to the right by a predesignated number of positions (I). Store the contents in location B.

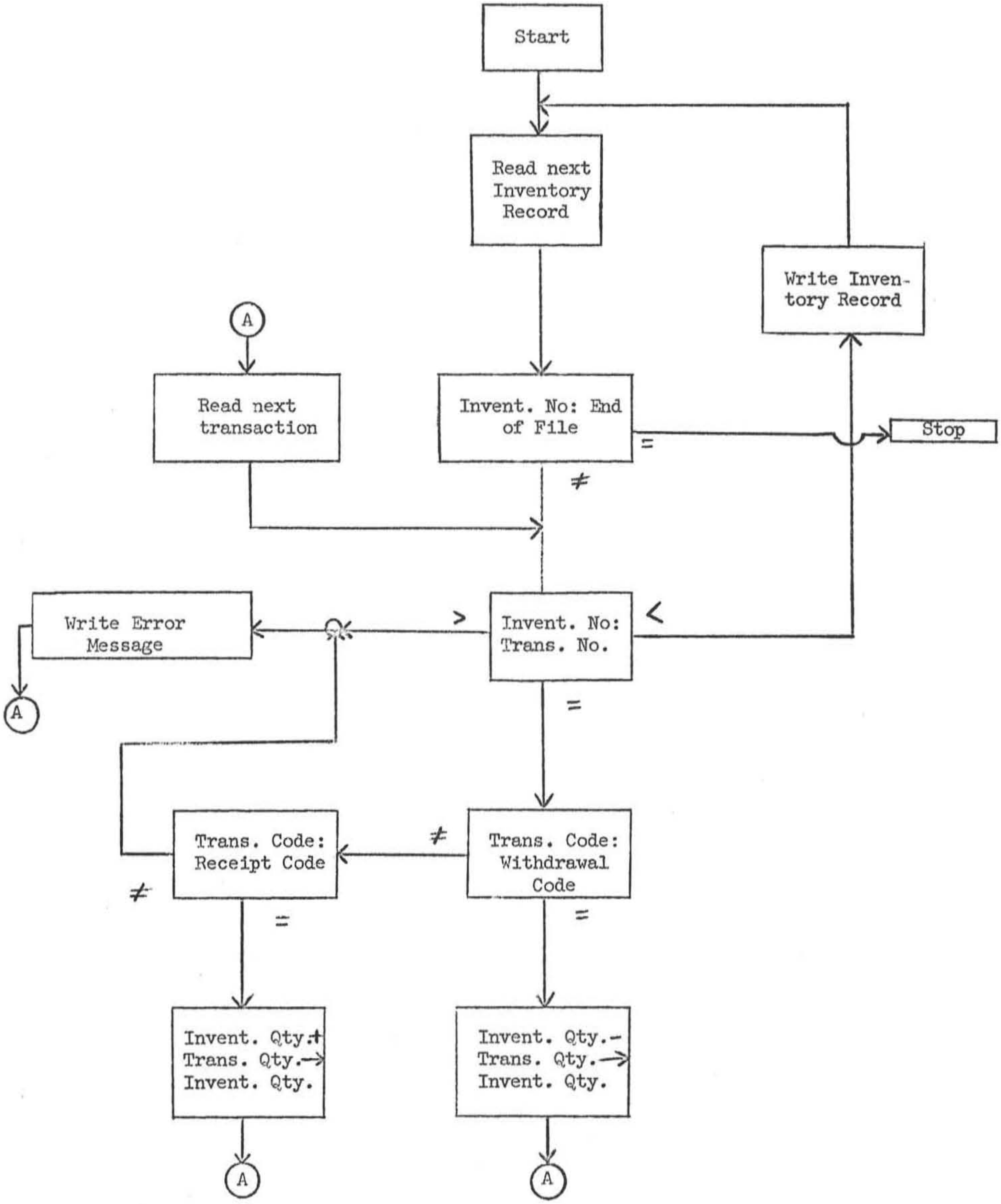
The same reasoning could be applied to shift left, shift right and round, shift left and test for overflow, etc. In arithmetic operations (and actually in any of the others) we can certainly consider the accumulator (s) as being merely special field locations so that there is no necessity for storing information in regular memory after an operation. For instance, MULT A, B where B was an accumulator address would simply mean to multiply the contents of the selected accumulator (B) by the contents of location A and store the result back in the selected accumulator (B).

5. Logical instructions -
any instruction which performs a normally defined Boolean function such as logical AND, logical INCLUSIVE OR, logical EXCLUSIVE OR, logical NOR, logical NOT, etc. The mode of operation would be similar to that of the arithmetic operations and could recognize both binary and unary logical instructions. Logical Not would be an example of a unary logical instruction. These could be used for control, masking, extracting, etc.

Another possibility for logical operations might include the presetting of either a logically true or logically false indicator which could be tested in a subsequent Branch instruction. One implication of this type of operation is that the computer should be capable of operation in a binary number mode (0,1), though this is not necessary for the other operations. It suggests some way of defining structure at the bit level rather than at the character level. Definition could be implicit in the instruction code itself; for example, regular arithmetic might always refer to a 4 bits per character construction; but logical operations might always use a one bit per character construction; but on move and compare operations character construction would not be significant except as required for partial word operands.

This list is not an attempt to be definitive nor are the suggested modes of operation necessarily optimal for a given class of problems. Nevertheless, I believe that they show the comprehensiveness and potential scope of a two-address logic as well as indicating the simplicity and ease with which many frequent business data processing operations could be handled. It is also obvious that any programming system constructed with this logic could provide for any of the modifications possible in a one-address or three-address language, including: indexing, partial word selection, debugging stops, etc.

To examine further some of the potential advantages and disadvantages of this approach we might review the following example which has been coded in each address mode. I have assumed a simple mnemonic instruction code set for each configuration. Except for initializing, ending, and handling transactions with identification numbers greater than the largest valid inventory number, the problem is flow charted as follows:



Start
Read next Inventory Record
Write Inventory Record
A
Read next transaction
Invent. No: End of File
=
Stop
≠
Write Error Message
>
Invent. No: Trans. No.
<
A
=
≠
Trans. Code: Receipt Code
Trans. Code: Withdrawal Code
≠
=
=
Invent. Qty.+ Trans. Qty.→ Invent. Qty.
Invent. Qty.− Trans. Qty.→ Invent. Qty.
A
A

Within the framework of the flow chart and except for the start and stop routines I have delineated one possible program for solving this problem on a one-address machine:

Address	Inst. Code	Field Address	Comments
01	Read	Input 1	Move next inventory record to input buffer
02	Relocate	Invent. work area	Move info. in input buffer to invent. work area
03	Bring	Invent. No.	Move invent. no. into accumulator
04	Compare	End of File No.	Test accum. vs. end of file no.
05	Branch Equal	Stop Routine	If set proper comparison indicators equal, comparison indicator is "on", Stop Routine address
06	Bring	Invent. No.	________________
07	Compare	Trans. No.	Test accum. vs. trans. no., etc.
08	Branch Greater	28	To Transaction Error Routine
09	Branch Smaller	19	To write inventory record routine
10	Bring	Trans. Code	________________
11	Compare	Withdrawal Code	________________
12	Branch Not Equal	22	To Receipt Test Routine
13	Bring	Invent. Qty.	________________
14	Subtract	Trans. Qty.	Subtract trans. qty. from accum. result in accum.
15	Store	Invent. Qty.	Store contents of accum. at Invent. Qty. Location
16	Read	Input 2	________________
17	Relocate	Trans Work Area	________________
18	Branch Uncond.	06	________________
19	Relocate	Invent. Work Area	________________
20	Write	Output 1	________________

Address	Inst. Code	Field Address	Comments
21	Branch Uncond.	01	
22	Compare	Receipt Code	
23	Branch Not Equal	28	To transaction Error Routine
24	Bring	Invent. Qty.	
25	Add	Trans. Qty.	
26	Store	Invent. Qty.	
27	Branch Uncond.	06	
28	Relocate	Trans. Work Area	
29	Write	Output 2	
30	Branch Uncond.	06	

On the basis of 2 decimal digits for the instruction code and 4 decimal digits for the address, this program would require 30 x 6 = 180 memory location units.

In a similar way, I have prepared a possible program for a three-address machine.

Address	Inst. Code	Field Address A	Field Add. B	Field Add. C
01	Read	Input 1		Invent. Work Area
02	Compare Equal	Invent. No.	End of File No.	Stop Routine
03	Compare smaller	Invent. No.	Trans. No.	09
04	Compare greater	Invent. No.	Trans. No.	14
05	Compare not equal	Trans. Code	WithdrawalCode	11
06	Subtract	Invent. Qty.	Trans. Qty.	Invent. Qty.
07	Read	Input 2	————	Trans. Work Area
08	Branch Uncond.	————	————	03
09	Write	Invent. Work Area	————	Output 1
10	Branch Uncond.	————	————	01

(Cont.)

Address	Inst. Code	Field Add. A.	Field Add. B	Field Add. C
11	Compare Not equal	Trans. code	Receipt code	14
12	Add	Invent. Qty.	Trans. Qty.	Invent. Qty.
13	Branch Uncond.	____________	____________	03
14	Write	Trans. Work Area	____________	Output 2
15	Branch Uncond.	____________	____________	03

On the basis of two decimal digits for the instruction code and four decimal digits for each of the addresses this program would take 15 x 14 = 210 memory location units.

For comparison, the same problem is programmed for a two-address machine:

Address	Instruction Code	Field Address A	Field Address B
01	Read	Input 1	Invent. work area
02	Compare	Invent. No.	End of file no.
03	Branch	Equal	Stop Routine
04	Compare	Invent. No.	Trans. No.
05	Branch	Smaller	12
06	Branch	Greater	18
07	Compare	Trans. Code	Withdrawal code
08	Branch	Not Equal	14
09	Subtract	Trans. Qty.	Invent. Qty.
10	Read	Input 2	Trans. work area
11	Branch	Uncond.	04
12	Write	Invent. work area	Output 1
13	Branch	Uncond.	01
14	Compare	Trans. Code	Receipt code
15	Branch	Not equal	18
16	Add	Trans. Qty.	Invent. Qty.
17	Branch	Uncond.	04

(cont.)

Address	Instruction Code	Field Address A	Field Address B
18	Write	Trans. work area	Output 2
19	Branch	Uncond.	04

On the basis of two decimal digits for the instruction code and four for each address, this program would require 19 x 10 = 190 memory location units.

Let's examine these programs at least superficially to draw some tentative conclusions.

1. There is no striking difference in memory space required for either of the three programming modes.

2. The three-address mode does well because of the arithmetic capability and the combined compare and branch instruction.

3. The one-address system is a little more difficult and time consuming to write and requires more words of instructions (though not necessarily more memory space, dependent on the internal word structure).

4. Two-address logic does very well on move type instructions (read, write) and on "add-memory" operations.

5. There were 13 different instruction codes used for the one-address program. The three-address system used nine different instruction codes while the two-address system used only six. This is not necessarily significant, but it may be indicative of a somewhat simpler instruction code structure.

6. Almost half (9) of the 19 instructions in the two-address system were branch instructions. Suppose it were possible to change the concept of the compare instruction so that a specific indicator was examined to see if it was on or off, and suppose that the "success" branch was always the next regular instruction while the "failure" branch was a fixed interval away; then it would be feasible to eliminate virtually every branch except where three or more alternate exits existed or where the branch was unconditional. In the program under

discussion, this would have eliminated four branch instructions. This may be feasible to accomplish through the two-dimensional programming approach. This aspect of success-failure physical location will be discussed in the next section.

On the basis of these comparisons, I believe it is evident that intensive study of two-address programming systems may offer important ways to reduce computer logic cost while providing more efficient programming instructions.

Section C- Controlled Two-Directional Branching

In normal programming methods with a one-address or multiple-address machine, the succeeding instruction in serial sequence is always implied as the alternate address on a branch instruction. The explicit branch is stated directly in the instruction. This is all that can be expected of any one-dimensional programming scheme.

In contrast, a two-dimensional programming system implies a two-dimensional branch. If the test succeeds, then the proper subsequent instruction follows next in the same row (or column). If the test fails, then the subsequent instruction is the first test in the next row (or column). Since the total number of columns per row is known, it is a straight-forward matter to compute the next instruction location for a test failure.

With this concept, we can think of a two-address instruction of "compare greater", which implicitly defines the success instruction address and the failure instruction address. This would require defining a complete set of the Compare instructions, which were of significance (greater, smaller, equal, not equal, greater or equal, smaller or equal).

Avoided would be the definition of any Branch instructions. Using this approach, the two-address program for the simple inventory problem used as an illustration could be reduced by eliminating 03, 05, 08, 15. However, it would require Compare Equal, Compare Smaller, Compare Greater and Compare Not Equal instructions. With this change, the program would be reduced to 15 instructions; with 10 characters each, 150 memory location units would be needed.

Section D - Relative Addressing and "Contained" Constants

All digital computers which have been announced up to now have had an instruction structure, which has called for stating one or more specific operand addresses. Some provision has been made for modification of these through the use of index registers, but the operating program presupposes absolute addresses.

In preparing new programming languages (symbolic assembly programs, compilers, etc.), one of the major efforts has been to enable the programmer to avoid this fixed address assignment. Two basic approaches have been taken to solving this problem. The first and less sophisticated, is to use a relative address like V014, which means the 14th word after that location designated as V000. This enables the program to be segmented, yet during the compiling stage, there is just the quite simple job of calculating the actual address of V014 as the location of V000 + 14. This is a very common practice. It does require, however, that the programmer in effect sub-structure the memory assignment and remember to use the correct relative address whenever he refers to that information.

A second approach has attempted to improve this area further. This is the concept adopted for FORTRAN and the Commercial Translator Language. Here, a mnemonic code name is assigned to the information field. For Example, EMPNO might refer to the Employer Pay Number field. FORTRAN restricts this to a six (6) character code. Commerical Translator allows the use of up to thirty characters plus adjectival modification to indicate file and record hierarchy. Because of the mnemonic aid,it is expected that the programmer will have far less trouble writing the correct pseudo-address, which the field name, of course, has become. During the compiling, each of these names will be assigned an actual address and each time the name occurs, this same actual address will be assigned. This is significantly easier for the programmer, particularly in explaining or communicating the program to someone else. It would also be a great aid in debugging except that the program debugs at the machine language level, which implies fixed addressing; this in turn means that the programmer has to convert from the absolute address to the field name that he has been using.

In preparing machine language programs, it has also been historically necessary to store any constants required and then call them out through using the appropriate absolute address. With relative addressing,the problem is only helped slightly since a memory location must still be used to store the needed constant. With FORTRAN, etc., the programmer may use a constant directly in his instructions; e.g. Y = 16 X. The compiler assigns this constant a location and, in the object program, refers to this location. While this is a big improvement for the programmer, it still uses up memory space for all the various constants needed. One interesting variation is to use the addresses themselves as constants; this yields the most commonly used integers. It is proposed in this paper that a new programming language be constructed so as to permit direct use of mnemonic addressing and so as to contain constants within the instruction word, thereby requiring no additional memory space for their storage.

(cont.)

It is further proposed that during compiling the computer may assign a relative address to each mnemonic address and that the internal computer logic should be structured so as to operate directly on this relative address. Furthermore, that the contained constants, where possible, be retained within the instruction and not independently stored. This would require that the instruction code recognize that one of the operands was a constant and not a mnemonic address.

If we consider these suggestions in context with the two-address logic recommendation, we can consider that the instruction structure for the programmer might appear as follows:

1. Move Instructions

   <u>Move</u> <u>(Field Name A)</u> to <u>(Field Name B)</u>

   or

   <u>Move</u> <u>Constant</u> to <u>(Field Name B)</u>

2. Relational Instructions

   <u>Compare</u> <u>(Field Name A)</u> with <u>(Field Name B)</u>

   or

   <u>Compare</u> <u>Constant</u> with <u>(Field Name B)</u>

3. Branch Instructions
   Not affected

4. Arithmetic Instructions

   <u>Add</u> <u>(Field Name A)</u> to <u>(Field Name B)</u>
   [and store in <u>(Field Name B)</u>]

   or

   <u>Add</u> <u>Constant</u> to <u>(Field Name B)</u>
   [and store in <u>(Field Name B)</u>]

Unary arithmetic would also have both configurations.

5. Logical Instructions
   Similar to the arithmetic operations.

During compiling, the only changes would be to convert the operation name to an operation code and translate the field names into relative addresses. For example:

	Move	(Field Name A)	to	(Field Name B)
might become	27	V001		V009
	Move	4728	to	(Field Name B)
might become	28	4728		V009

The major reason for converting to a relative address is that current-day machines lose significant time in performing a dictionary look-up operation. It is quite conceivable, of course, that new machine components may change this picture considerably in which case it might be advantageous to store the mnemonic address rather than a relative location.

A major concern, which has been alluded to earlier is the difficulty of debugging in a machine oriented language. It is therefore believed worth considering the preparation of interpretive programs, which allow the computer to debug in the systems oriented language itself. Once the program has run satisfactorily, then the regular compiler could be used to prepare a set of instructions suitable to the particular computer. To generalize this point it would, for example, seem worthwhile to construct an interpretive FORTRAN processor which would permit direct debugging in the FORTRAN language, even though eventually an objectprogram will be compiled. Such programs should be relatively inexpensive to prepare and will increase significantly the desire of many experienced programmers to use these advanced programming languages. This concept is tied somewhat to the idea of relative addressing and contained constants, since with these two tools the interpretive program can be quite simple and the analysis of results quite straight-forward.

Section E - A Suggested Minimum Two-Dimensional Language

For the purpose of establishing a common frame of reference, a "minimum" language is described, which would enable a computer to rapidly execute the bulk of the operations, which seem to be required in two-dimensional programming. There is obviously nothing magical about this particular set of instructions. However, industrial computing experience indicates that these would permit a one to one translation of many relations and actions into computer instructions. The original set has intentionally omitted indexing, partial or multiple word operations, and logical (Boolean) manipulations. These items are discussed later in this section. In the symbolic statements included after each instruction definition, () means "contents of location designated" and $\rightarrow$ means "replaces".

Move Instructions

READ A, B

Move the next record from input source A to a series of internal locations beginning with location B. Source A may be a card reader, punched tape reader or any selected magnetic tape transport. This instruction could operate with fixed record length (N words or characters or bits), variable record length (separated by a recognizable record mark), or with defined record length as a modifier to the READ instruction (such as Read 60 to imply Read 60 consecutive words).

$$(A) \rightarrow (B)$$

WRITE A, B

Move the next record from a series of internal locations beginning with location A to output destination B. Destination B may be a card punch, tape punch, printer or any selected magnetic tape transport. The same statements about record length that were made in the READ instruction would apply to the WRITE instruction.

$$(A) \rightarrow (B)$$

ASSIGN A, B

Move the contents of the Field A designated location to the Field B designated location; this instruction moves information from one memory location to another memory location.

$$(A) \rightarrow (B)$$

ASSIGN CONSTANT A, B

Move the contents of Field A to the Field B designated location; this instruction transfers a constant to a memory location.

$$A \rightarrow (B)$$

(cont.)

COMPARE EQUAL A, B

Compare the contents of the Field B designated location with the contents of the field A designated location. If these are identically equal, then branch to the preassigned success location (S); if these are not identically equal, then branch to the preassigned failure location (F).

if $(B) = (A)$ then next instruction at S

if $(B) = (A)$ then next instruction at F

COMPARE SMALLER A, B

if $(B) < (A)$ then go to S

if $(B) \geqslant (A)$ then go to F

COMPARE GREATER A, B

if $(B) > (A)$ then go to S

if $(B) \leqslant (A)$ then go to F

COMPARE NOT EQUAL A, B

if $(B) \neq (A)$ then go to S

if $(B) = (A)$ then go to F

If the machine's internal characteristics lend themselves to a slightly more elaborate mode of operation, then the instruction set may also include:

COMPARE SMALLER OR EQUAL

COMPARE GREATER OR EQUAL

COMPARE CONSTANT EQUAL A, B

Compare the contents of the Field B designated location with the contents of Field A. If these are identically equal, then branch to the preassigned success location (S); if these are not identically equal then branch to the preassigned failure location (F).

if $(B) = A$ then go to S

if $(B) \neq A$ then go to F

COMPARE CONSTANT SMALLER A, B

if $(B) < A$ then go to S

if $(B) \geqslant A$ then go to F

COMPARE CONSTANT GREATER A, B

if $(B) > A$ then go to S

if $(B) \leq A$ then go to F

COMPARE CONSTANT NOT EQUAL A, B

if $(B) \neq A$ then go to S

if $(B) = A$ then go to F

If the internal machine characteristics lend themselves to this mode of operation, the instruction set may also include:

COMPARE CONSTANT SMALLER or EQUAL

COMPARE CONSTANT GREATER or EQUAL

GO TO A

transfer program control to location designated in Field A. This is an unconditional branch instruction. The reason it is needed rests with the limitations of a two-dimensional programming system (in contrast to an n - dimensional system).

Arithmetic Instructions

ADD A,B

Add the contents of the Field B designated location to the contents of the Field A designated location. Store the result in the Field B designated location. Either Field A or Field B may be a special high-speed accumulator.

$(B) + (A) \rightarrow (B)$

SUBTRACT A, B

$(B) - (A) \rightarrow (B)$

MULTIPLY A, B

$(B) \times (A) \rightarrow (B)$

DIVIDE A, B

$(B) \div (A) \rightarrow (B)$

We have omitted in this instruction set exponentiation or unary arithmetic operations. In the problems which we have handled to date, their frequency of use has been such that this need could be best served through use of stored subroutines.

ADD CONSTANT A, B

Add the contents of the Field B designated location to the contents of Field A. Store the result in the Field B designated location.

$(B) + A \rightarrow (B)$

SUBTRACT CONSTANT A, B

$(B) - A \rightarrow (B)$

MULTIPLY CONSTANT A, B

$(B) \times A \rightarrow (B)$

DIVIDE CONSTANT A, B

$(B) \div A \rightarrow (B)$

SHIFT LEFT A, B

Shift the contents of the Field B designated location to the left by A positions.Conceptually this operates on a predefined word length. The result is stored in the Field B designated location. If there is an overflow consider this as a failure; if no overflow, this is a success.

SHIFT RIGHT A, B

Shift the contents of the Field B designated location to the right by A positions. The result is stored in the Field B designated location.

If desired, a SHIFT RIGHT and ROUND instruction may be included.

Logical (Boolean) Instructions have been omitted.

Miscellaneous Instructions

NO OPERATION

STOP

Simple Extensions of the Minimum Language

Among additional features which may be desirable in a two-dimensional programming system are: address indexing; partial word movement; multiple word movement; error control (debugging stops and input variance detection); extended arithmetic capability including square root, integral exponentiation (square and cube) and certain trigonometric functions (sin, tan). A simple approach to some of these features is described in this section.

One problem in address indexing has been the attempt to consider multiple subscripts. Writing a compiler to automatically recognize and handle multiple subscripts is a relatively complex chore. A simple concept which handles any number of subscripts is through a calculated index which is the function of the subscript values.

INDEX ASSIGN I,A,B

> Modify the Field A designated location by the contents of the Field I designated location. Move the contents of this modified Field A designated location to the Field B designated location.

$$(A + (I)) \Rightarrow (B)$$

ASSIGN INDEX I,A,B

> Modify the Field B designated location by the contents of the Field I designated location. Move the contents of the Field A designated location to the modified Field B designated location.

$$(A) \rightarrow (B + (I))$$

The contents of the Field I designated location can, of course, be previously defined by any acceptable operation. Let's assume we have a three dimensional matrix stored in consecutive locations first by column, then by row, then by table. There are three parameters to be defined: R (max Rows), C (max Columns), T (max Tables). We will define r as the row subscript, c as the column subscript, and t as the table subscript.

$$I = f\ (r,c,t)$$

$$I = \text{Initial Location} + r + cR + tRC$$

Test for completion would be I versus Initial Location $+ R\ (1+C(1+T))$. This principle can, of course, be extended to any n-dimensional subscripting plan. It could, of course, also be applied in concept to any arithmetic or comparison operations.

Partial word movement could be handled through a similar approach:

PARTIAL ASSIGN I,F,A,B

Move the Ith (Initial) through the Fth (Final) position of the contents of the Field A designated location into the Field B designated location starting at the left-most position.

ASSIGN PARTIAL I, F, A, B

> Move the contents of the Field A designated location starting from the left-most position into the Ith through the Fth position of the Field B designated location.

Multiple word movement could be handled as follows:

MULTIPLE ASSIGN M, A, B

> Move M words starting at the Field A designated location into a series M words beginning at the field B designated location.

Error Control for debugging stops and input variance detection could be handled as follows:

ERROR GO TO A, B

> Transfer control to the Field A designated location. Set up to return control to the Field B designated location upon completion of processing the next table.

Unary Arithmetic could follow the following form:

SQRT A, B

> Find the square root of the contents of the Field A designated location; store the result in the Field B designated location.

Recommendations

It seems evident that present approaches to systems-oriented languages do not appear to be capable of making a basic break-through in the one really critical programming problem: Systems description. Until a technique is developed which supersedes flow charting and yet is readily computer-understandable we shall not have achieved an effective application language.

Because the logic of two-dimensional programming seems irrefutable from a users standpoint it certainly seems worthwhile to aggressively pursue a research and development program aimed at exploring and advising techniques in this general area. The program below would, I believe, have a reasonably good chance at full success:

(1) Propose a specific two-dimensional language based on the Hunt Foods', Aeronutronics and IBM work (particularly the direction suggested by Perry Crawford). The language may differ for various types of usage (Input, Output, File Maintenance, Decision-making, Arithmetic, etc.)

(2) Prepare a simple interpretive program to solve two-dimensional programs. Also prepare a converter for translating from a Commercial Translator level of word choice to a machine-oriented level.

(3) Program a variety of problems using the language. Try to teach the language to non-programming personnel; have them write programs for areas of knowledge using the language.

(4) Run a few trial problems with the programs written in the two-dimensional language. Explore techniques for debugging and program modification.

(5) Revise the language; this may include permitting, but not requiring, certain desirable special features like SORT, UPDATE, MERGE.

(6) Prepare a "bootstrapped" compiler for two or more different computers. This will give a deeper insight into the language structure. Prepare a new Interpreter and Converter.

(7) Conduct further experimentation bringing in appropriate Field Sales personnel.

(8) Prepare manuals on the language covering the following areas:

   (1) primer
   (2) reference
   (3) application experience
   (4) Interpretive, Converter, Compiler Programs.

(9) Publish the results and present to CODASYL or other appropriate professional groups.

This is obviously an ambitious program and would probably involve 3-5 full time people together with appropriate help from many others on a part-time basis. Because the need is so great, I feel that the time schedule should be intentionally brief with completion targeted at 12-18 months from initiation.

I would estimate that the total cost of such a project including computer time, programming, technical writing, outside consultants, office support, and salaries for full-time personnel (but not for part-time) would approximate $300,000.

If the project worked out as I would hope then the reward should be a major advance in the programming (in contrast to the coding) art. We would have a basic new tool for systems design and a firm basis for language standardization.

USING DECISION TABLES
FOR
PRODUCT DESIGN ENGINEERING

By Burton Grad
Manager
Systems Engineering Services
International Business Machines
Corporation

Prepared for AIEE Winter Conference
January 24, 1962
New York City, New York

A certain computer manufacturer has a variety of machines in its product line. The 1401, a solid state machine, has many thousands of units on order. A large-scale binary machine, the 7090, uses the same instruction set as the 704, which is no longer in production. The 7080, a logical successor to the 705 and 702 can use up to ten advanced disc memory units besides magnetic tape units. Already announced, but not yet delivered is the 1410, a medium scale machine compatible with the small scale 1401.

Now with this type of information what are the characteristic values for each of the seven machines mentioned? In narrative form, of course, the information is disjointed, probably incomplete, and certainly not concise. Tabular form, with its clear structure suggests itself as one means of effectively organizing this data.

To construct a table from the information in this narrative, we first identify the names of the characteristics:

Machine identification
Order statistics
Solid state
Instruction logic
Disc memory units
Magnetic tape units
Scale of machine
Type of machine

Then we fill in the values described for each characteristic:

Machine Ident.	1401	7090	704	7080	705	702	1410
Order statistics	1000's on order		not in pro-duction				not yet del'd.
Solid state	yes						
Instruction logic		same as 704		same as 705	same as 702		same as 1401
Disc memory units				up to 10 advanced			
Magnetic tape units				yes			
Scale of machine	small	large					medium
Type of machine		binary					

This tabular arrangement restores order to the information while conserving space, displaying meaningful relationships, and explicitly indicating missing values.

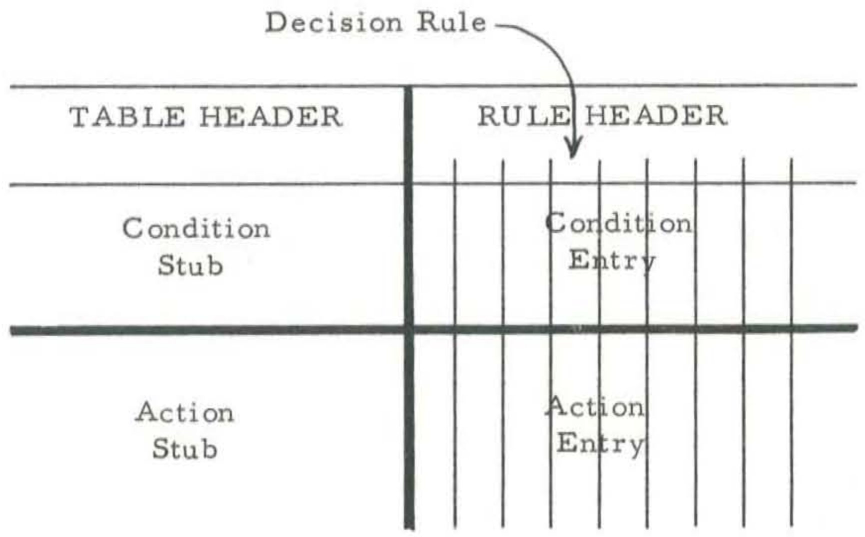
Although tabular form recording has been known and practiced for hundreds and even thousands of years it has been used mainly for summarizing facts and previous experience in statistical tables and accounting reports. Subsequent to the development of Boolean Algebra came truth tables, using tabular form for analyzing logical implications. More recently tables have been used instead of narrative, flow charts, and logical equations to express complex decision logic. These are called decision tables to connote the subject and the form -- tabular form for recording decision logic. In the more elemental examples a decision table strongly resembles a statistical or financial table; consequently the form and terminology of those tables has been adopted.

The following insurance premium decision table illustrates the pertinent features:

Health	Excellent	Excellent		Poor
Age	$\geq 25, < 35$	$\geq 25, < 35$		≥ 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

Upon examination, the decision table reveals that insurance premium rate and policy limit are a function of health, age, section of country, and sex. If the applicant is in excellent health, between 25 and 35 years of age, from the East, and is a male, then his rate is $1.27, and the insurance limit is $200,000. All of the other alternatives are then clearly set forth, one by one, across the table.

To clearly show the essential elements of a decision table, its basic arrangement and terminology may be outlined as follows:



The heavy lines serve as demarcation: CONDITIONS are shown above the heavy horizontal line, ACTIONS below. The STUB is to the left of the heavy vertical line, ENTRIES to the right. A condition states a relationship. An action states a command. If all the conditions in a column are satisfied then the actions in that column are executed. Each such vertical combination of conditions and actions is called a RULE. In the same column with the entries for each rule, there may be specialized data relating to that rule; this is called the RULE HEADER. Similarly, each table may have certain specialized information which is called the TABLE HEADER.

Using this general structure, an increasing number of companies have begun to state complex logical processes in decision tables. Since product design engineering is a complex logical process, it seems reasonable to investigate this area for the possible use of decision tables.

The design engineer has the task of determining product characteristics as a function of customer specifications. With all the help of previous designs, standard models, and so on, he is still frequently required to apply certain rules and formulas to develop special models or variations to meet special customer requirements.

The basic application of decision tables would be to record design engineering logic so completely and accurately that, given a set of specifications, a correct set of product characteristics could be generated. Such an engineering decision table would look like this:

Customer Specification Names	Specification Values and Ranges
Product Characteristics Names	Characteristic Values and Formulas

The design of a product like a switchboard instrument might involve 100 different specifications, 500 variable characteristics, with the logic expressed in 100 decision tables of 10 rules and 10 rows each.

To illustrate the use of decision tables, an armature coil of a hypothetical switchboard instrument will be used. The customer specifications include possible values or ranges.

Customer Specification Names	Specification Values and Ranges
Service	"AC" or "DC"
Application	"Temperature" or "Speed"
Rating Units	"MV" or "MA"
Rating Value	10-900
Number of Phases	1 - 3
Scale Size (in.)	4 or 8

The product characteristics are:

Number of windings
Main Winding Number of Turns
Damper Winding Number of Turns
Main Winding Wire Material
Damper Winding Wire Material
Main Winding Wire Diameter (mils)
Damper Winding Wire Diameter (mils)
Main Winding Number of Layers

The first decision table develops certain common data (intermediate values) which are needed for subsequent decision tables.

Table #1

Rule No.	1	2	3	4
Service	"DC"	"DC"	"AC"	"AC"
Application	"Temperature"	"Speed"		
Rating Units	"MV"	"MV"	"MV"	"MA"
Number of Phases	1	1	1	≥1
Type of Armature	"Moving Coil"	"Moving Coil"	"Electro Dynamic"	"Inductive"
Number of Windings	1	2	2	1 + Number of Phases
Next Table	2	2	2	2

Rule 3 reads: <u>if</u> the Service is "AC", <u>and</u> the Rating Units is "MV", <u>and</u> the Number of Phases is 1, <u>then</u> the Type of Armature is "Electro Dynamic", <u>and</u> the Number of Windings is 2, <u>and</u> Go To Table #2.

Notice particularly four features which give decision tables potency far beyond the traditional truth table:

(1) "Not pertinent" conditions can be ignored as shown by the blanks for Application in rules 3 and 4.
(2) Value limits can be used rather than each individual value such as in rule 4: Number of Phases greater than or equal to 1.
(3) Formulas can be used to give many results where a mathematical pattern exists as shown in Number of Phases in rule 4.
(4) Actual values are either numeric or they are shown inside quote marks as "Moving Coil".

Now turn to Table #2; every armature coil has a main winding; consequently its characteristics are determined first. These are all distinguished by an MW for Main Winding.

Table #2 - Next Table #3

Rule No.	1	2	3
Type of Armature Rating Units Rating Value	"Moving Coil" "MV" 10-75	"Moving Coil" "MV" 76-200	"Electro Dynamic"
MW Number of Turns MW Wire Material MW Wire Diameter MW Number of Layers	52 "Alum" 16 4	13 "Alum" 16 1	Formula 5 "Cu" 4 MW Number of Turns/26

Rule 2 reads: if the Type of Armature is "Moving Coil", and Rating Units is "MV", and Rating Value is between 76 and 200, then the Number of Turns is 13, and Wire Material is "Alum", and Wire Diameter is 16 mils, and Number of Layers is 1, and go to Table #3 next.

Features shown in this decision table include:

(1) Direct indication of value range as in Rating Value.
(2) Use of a formula name as for Number of Turns.
(3) Next Table indication in the table header.

The final table covers the key characteristics for the damper winding.

Table #3

Rule No.	1	2	3
Scale Size Rating Units Rating Value Number of Windings	4 "MV" 10-75 ≥ 2	4 "MV" 76-200 ≥ 2	8 "MV" 10-200 ≥ 2
DW Number of Turns DW Wire Material DW Wire Diameter	24 "Cu" 16	45 "Cu" 8	.5* MW Number of Turns "Alum" 12

Out of necessity this instrument armature coil example was rather briefly outlined; it is intended only to convey the concept of tabular form as applied to one small part of the design engineering process. To solidify the meaning and use of decision tables the reader may try a simple example:

Determine the Armature Coil characteristics given the following customer specifications:

Service - "DC"
Application - "Speed"
Rating Units - "MV"
Rating Value - 100
Number of Phases - 1
Scale Size - 8

In Table #1 we find that Rule 2 is satisfied:

Type of Armature - "Moving Coil"
Number of Windings - 2

Then we go to Table #2 where we determine that Rule 2 is the proper one:

MW Number of Turns	= 13
MW Wire Material	= "Alum"
MW Wire Diameter	= 16
MW Number of Layers	= 1

Then in Table #3 we find that Rule #3 is the correct one:

DW Number of Turns	= 7
DW Wire Material	= "Alum"
DW Wire Diameter	= 12

Without much argument, design engineering is one of the most appealing and intriguing of the many varied applications for decision tables. Here the potential of the decision table is not just in the displacement of routine engineering effort, but in the significant and challenging task of improving product quality and performance while lowering cost.

For the first time, the inherent design logic of a product can be unlocked from musty files of blueprints, bills of material, and engineering instructions. It can then be readily communicated to succeeding functions of the business with a consequent cascading of its benefits to all areas. With this sort of contribution to make, the decision table, as a method for displaying product design logic stimulates better product design--in less time--at lower cost.

Progress in Decision Table Applications

by

Thomas B. Glans
International Business Machines Corporation
White Plains, New York

November, 1961

Progress in Decision Table Applications

by Thomas B. Glans
IBM

Decision tables have attracted widespread interest in the relatively short time since the first significant experiments were announced. Experience now indicates that they often have clear-cut advantages over other techniques in three rather diverse areas: systems analysis, documentation, and programming.

In this paper we display the concept of a decision table, cite several successful experiments, and suggest a course of action for further exploration of the advantages offered by decision tables.

THE TABLE CONCEPT

The basic idea of how a decision table is organized and used is fairly familiar to many, but perhaps not to all. Examples will suffice to demonstrate the principal concepts.

Figure 1 is part of a table defining the rate and policy limit of an insurance company, as a function of age, health, and section of country.

	Rule 1	Rule 2		Rule 30
Age	≥ 25 < 35	≥ 25 < 35		≥ 65
Health	Excellent	Excellent		Poor
Section of Country	East	West		West
Rate/1000	1.57	1.72		5.92
Policy Limit	200,000	200,000		20,000

Figure 1

The first decision rule can be paraphrased as follows: If age is greater than or equal to 25 and less than 35, and health is excellent, and section of country is East, then the rate per thousand is \$1.57 and the policy limit is \$200,000. The underlined words are implied by the table structure. The other rules are alternatives to this one; thus it does not matter which rule is examined first: only one rule can be satisfied in a single pass through this decision table.

The information in Figure 1 is shown in an exploded view in Figure 2, indicating more clearly the parts of a table and the terms that are used to describe them. The double lines serve to separate conditions above the horizontal double line from actions below, and to separate the stub to the left of the vertical double line from entries on the right. The essential nomenclature is completed by adding at the top a title section, called a table header, and by adding a rule header over the entries.

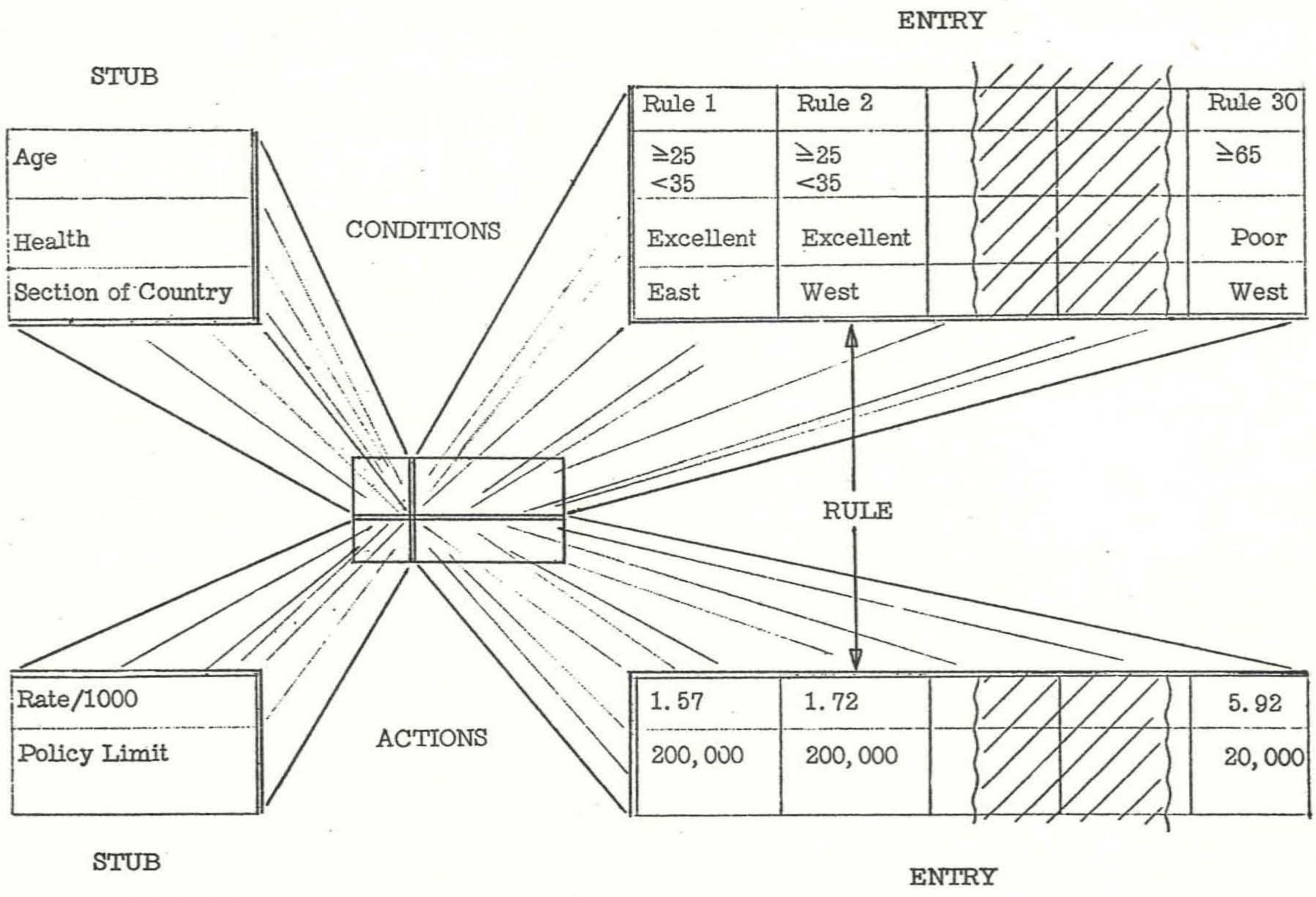


Figure 2. Exploded view of the table of Figure 1.

Tables may be used in a slightly different form as shown in Figure 3, which is a table documenting the decision logic for a charge desk at a department store.

TABLE: CREDIT	Rule 1	Rule 2	Rule 3	Rule 4
Credit limit is o.k.	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

Figure 3

The first rule reads: <u>If</u> credit limit is OK <u>then</u> approve the order. The second rule reads: <u>If</u> credit limit is not OK <u>and</u> pay experience is favorable <u>then</u> approve the order. The other two rules can be read just as easily as these.

<u>APPLYING DECISION TABLES</u>

Decision tables seem to offer significant advantages in three areas: analysis, documentation, and programming.

They were originally developed for system analysis, as an alternative to flow charting. In addition to aiding in effective analysis decision tables also proved useful for communicating the system logic thus, serving as a documentation technique. Programmers, seeing the form, began using it for writing source programs, recognizing that with the clear structure, a compiler could be readily prepared.

A few examples of the use of tables in the areas of analysis, documentation, and programming will indicate how successful field experiments have been.

Analysis

Mr. Leo O'Leary, a member of IBM's Applied Programming staff, was faced with a difficult logical problem in the data division of one of IBM's COBOL processors. He was reluctant to use normal techniques for doing this analysis and programming job because of its complexity. He felt he needed a powerful organizational tool in order to see clearly the numerous alternatives. He had no extensive experience with the use of decision tables, but he had attended a one-day seminar on the subject and decided to give it a try.

After some preliminary study he discovered that there were only seven independent conditions in the problem, and that each of them was of a binary nature. He mechanically wrote down all 128 combinations of these seven variables in the form of a table. He then proceeded to discard the combinations that were impossible because of the construction of the language or because they had been handled by previous processing runs. During this process he discovered a number of cases that called for the same actions; these he grouped together by using "not pertinent" entries in the table. For each rule he wrote the actions required and discovered, somewhat to his surprise, that there were only 20 distinct actions required. The completed table had 70 rules and 20 actions; by the way the table had been constructed, he was certain that all relevant combinations had been handled properly. This table provided the basis for a very simple transition to machine code, in a manner that is quite interesting in itself but unfortunately not relevant here. The soundness of the whole approach was completely justified by the speed with which the resulting program was checked out.

The original space estimate for this job had been 1200-1500 positions; the completed program took 530. Analysis and machine coding required four days, and the job was checked out in just one day.

As an analysis technique they seem to be more manageable than flow charts or narrative descriptions. They present logical alternatives in an easily-understood graphical form. Since the alternatives are so clearly displayed, they are easy to check for completeness and consistency, leading to thoroughness and accuracy.

Documentation

Mr. John Czerkies, of IBM Corporate Data Processing, had just been informed that one of his programmers was to be promoted and would be leaving within a week. The programmer had just finished his seventh program for their 1401; none of these programs had yet been documented to the point where another programmer could take over. While Mr. Czerkies first considered asking the programmer to prepare the usual flow charts, he rejected this possibility because he estimated the job would take three weeks. He had just heard a talk on decision tables and while he had no experience with them he wondered whether they might not provide a solution to his problem. He spent one hour with the programmer, instructing him in decision table technique and selecting a particular table format. Mr. Burton Grad, one of the developers of the decision table concept, spent half an hour with them to check out their approach, which seemed good.

The programmer than went to work. In just <u>two days</u> he wrote the decision tables necessary to display the logic of all seven programs. Since then, the programs have required modification and correction. The programmer taking over, using just the decision tables, in conjunction with the machine listings has had no trouble making the changes.

To indicate how effectively a decision table can document system logic, consider the common file maintenance problem: processing a detail file against a master file, both in sequence on identification number. Special actions are required at the beginning and end of each file, and the various combinations of high, low, or equal must be taken into account. Figure 4 is a decision table showing the logic of this problem.

				Rule #				
TABLE: Update	01	02	03	04	05	06	07	08
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

Rule 1 states the starting condition. Rule 2 handles the end of job conditions. Rule 3 describes the situation when the end of detail has been reached, but not the end of master. Since there can be no further changes, additions, or deletions to the original master, the actions are to write the updated master from the master area, read another master, and then return to the beginning of the table.

In Rule 4, the end of master has been found, but not the end of detail. Rules 5, 6, and 7 are concerned with cases where neither the detail nor the master file has ended. Rule 5 considers the event when the detail is less than the master. In Rule 6 the detail is greater than the master. Rule 7 covers the case where master and detail are equal.

The final rule, Rule 8, is the ELSE situation. When this occurs something has gone wrong, since all legitimate possibilities have already been examined. Rule 8 will take care of cases involving sequence errors in the master file and certain types of sequence errors in the detail file.

This example shows clearly how a decision table may be valuable for system documentation. Decision logic cannot be presented as concisely with a block diagram. Drawing a block diagram would take longer than developing this table. There is a much higher probability that the block diagram would contain logic errors and omissions. Decision tables can be read by other people with a minimum of training and explanation.

Programming

Eastman Kodak has used tables extensively in many areas of data processing. For purposes of comparing tabular-form programming with conventional methods, two similar problems were prepared by equally experienced programmers. Mr. E. O. Althoff, of the Eastman Kodak data processing staff, provides the time comparison shown in Figure 5.

Autocoder-level Programming		Tabular-form Programming	
Analysis	80 hours	40	
Learning	40	21	
Flow Charting	62	44	*Includes time to write an interpretive table processing routine.
Coding	136	39	
Testing	174	54	
	492	198*	

Figure 5. Comparison of times for programming two equivalent jobs using standard programming methods and tabular programming.

A number of additional experiments have produced similar savings. Based upon this, Mr. Althoff indicates that future work in his group will utilize tabular-form programming as a regular practice.

Mr. Harry Cantrell, Manager, Data Processing for the Large Steam Turbine-Generator Department of the General Electric Company, after careful investigation has instructed his computer people to use decision tables as their basic programming system.

Mr. Cantrell indicates that they have completed twenty-five 704 applications and are extremely enthusiastic about the savings in time that have resulted. They are getting 40 to 80 checked out program steps per hour.

As a programming method, decision tables used with a suitable language, reduce program definition, coding, and debugging time. By rearranging the rules in a table, efficient programs can be produced. Tables also provide a natural method of segmenting the program leading to easier debugging and maintenance.

THE IBM 1401 TABULAR PROGRAMMING SYSTEM

During the last year IBM has developed an experimental tabular programming system including a processor to compile source program tables into machine language instructions. This system is currently designed to run on a 4000 character 1401 with a 1405 (RAMAC® disk file).

The language is built on the foundation of the decision table concepts presented earlier. Figure 6 shows a typical 1401 decision table.

OP	NAME	OP	NAME	OP	NAME	OP	NAME	OP	NAME	OP	NAME
	AGE			LE	25	LE	25	GR	25	GR	25
	SEX	EQ			'M'		'F'				
	ACDNTS							EQ	0	GR	0
SET	RATE			+	RSKFAC			-	SPRFAC		
SET	PURATE	EQ			RATE		RATE		RATE		RATE
WRITE	RATECD			X		X		X		X	

Figure 6

Rule 1 reads: If age is less than or equal to 25 and sex is male, regardless of number of accidents then increment rate by the risk factor and set punch rate equal to rate and write a rate card.

This table illustrates the use of abbreviations for relational operators (LE, EQ, GR); the ability to include literals; a few of the English language action operators (SET...EQ, WRITE); the ability to specify two-address arithmetic in the body of a procedure table; and the ability to mix extended entry, where the actions "extend" into the action area, with limited entry, where the entire action is written in the stub and X's are placed in the entry portion.

Some of the other features of the language may be listed in summary fashion. A data description table is used to specify the characteristics of files, records, and fields. It may also contain specifications for constants and autopoint arithmetic computations, in a simple FORTRAN-like form. The full editing power of the 1401 is easily available. Word marks in the 1401 are conveniently handled by a LAYOUT operator that resets word marks in accordance with a designated record format; if a file has only one record format, no attention to word marks is required. Tables may be executed on a "DO" basis from other tables, allowing closed subroutines. The sequence of table execution is controlled by GO TO action commands. Extremely large programs can be written, since the tables are stored in 1405 RAMAC and brought to core storage as needed. Because of this automatic program storage allocation feature, the programmer need not be concerned about program storage problems--even in a machine with small core capacity.

At this time (November 10, 1961), the 1401 Tabular Programming System is being evaluated in the field. The results of this work will be included in the final version of the paper and discussed in the oral presentation at the Conference.

SUMMARY

There has been much progress made in the short time since decision tables were first publicized. Widespread field experiments have demonstrated their strength in analysis, documentation, and programming.

We feel that decision tables deserve even wider study and application, and that such work will lead to significant improvements in the power and applicability of the technique. In particular, we would like to emphasize that the structure of a decision table is distinct from the language used within it. Any suitable language can be used. Furthermore, the table structure concept is distinct from the characteristics of any larger programming system of which it may be a part.

We feel that the time is ripe for full-scale investigation of how decision tables can best be combined, in many different ways, with other types of programming systems. We do not propose tables as the replacement for all other systems, but we do feel that tables can be used with other concepts -- perhaps in ways not now envisioned -- to produce a combination much more powerful than any single part.

IBM 3

Originator — Retain Copy 2 and forward balance of set, with carbons intact, to the addressee.
Addressee — Prepare reply, detach remaining copies, and send Copy 3 to the return addressee.

Inter-Office Memorandum

RETURN TO

To: Mr B Grad	Dept. 797 -	Bldg. TV 454
Location: Dir of Dev - Sci / Cross Ind Appl		
Subject and/or Reference: DTAP - Decision Table Processor		

From: [illegible]	Dept.	Bldg.
Location or U.S. mail address (if external mail): Terre Haute Co		
Date: 4/5/72	Tie Line/Tel. Ext.	If no tie line note Area Code & Tel. No.

I want to let you know that I have been promoted to Education Development Rep, DPD Education Development, Raleigh N.C. I can be reached at 919-755-5366. I hope this will help speed our communication. I want to thank you for your interest. I hope our efforts bear fruit. Al

Reply

Date

Also to Cliff Burwick
2800 Sand Hill Road
Menlo Park Calif

M02-3127-13

September 19, 1972
Computer-Assisted Problem Solving
1501 California Avenue, Palo Alto, Calif. 94304
69M/2800 SHR
8/544-4301

SEP 22 1972

File
Dec Tables

Decision Table Processor (DATP)

Presentation Material on DATP by Dave Pfuetze

J.F. Bult

The reference document describing the Decision Table processor was submitted to David Low of the L.A. Scientific Center for evaluation. His conclusions are summarized in the following:

> The general idea is good. However, certain points need clarification, like the testing procedures. The use of a 3270 is desirable, but technical difficulties in the design and implementation must be expected in the presence of tables larger than the size of the screen. It is felt that the table entries should not be restricted to YES/NO values and should allow for numeric values.
>
> The resource requirements seem to be seriously underestimated. In particular, the definition of a language is not an easy task.
>
> D. Low recommends that a feasibility study, if any, be based on the use of an existing interpretive language (e.g., APL) so as to cut down the cost of the study by skipping the translation part of the project.

I recommend that the L.A. Scientific Center (D.Low) be associated with any study or development of this project, since knowledge and experience in the subject matter reside there, while no such talent can be found in Development.

[illegible]

A.A. Dubrulle

AAD:kep
cc: Mr. B. Grad
    Mr. D. Pfuetze (Raleigh)