SYSTEMS ENGINEERING SERVICES

CLEARINGHOUSE REPORT

PRELIMINARY REFERENCE MANUAL DRAFT

TABSOL - 225

A Tabular Systems Oriented Language
for the
GE 225 Information Processing System

February 1, 1961
Ref. No. 1B2

D. Klick
Computer Department
General Electric Company

INTERNATIONAL BUSINESS MACHINES CORPORATION

White Plains, New York

This material is distributed to keep IBM personnel informed of new developments. Selection is based on interest; this department makes no claim for the desirability of this approach nor necessarily recommends its use.

If additional copies are desired, please contact the Clearinghouse. No part of this material should be reproduced or distributed outside IBM without approval of the Clearinghouse.

PRELIMINARY REFERENCE MANUAL

TABSOL-225 --- A Tabular Systems Oriented Language

for the

GE225 Information Processing System

Computer Department
Applications Section
Programming Research and Development
Phoenix, Arizona

December, 1960

This document is a draft of a Preliminary Reference Manual and a language specification for integrating decision tables with the General Compiler. The information contained herein assumes a basic knowledge of computers and electronic data processing applications. Therefore, the manual should be used not as a text book but rather to augment already realized skills. Minor changes in language specification may occur during the implementation period of the compiler. Any changes that are made will be reflected in future and more final versions of this manual or in supporting material issued during the interim of implementation.

D. Klick
Programming Research

I. INTRODUCTION

Early automatic coding systems, such as assembly programs, employed mnemonic abbreviations in place of the computer's numerical instruction code and symbolic addresses in place of actual memory addresses. In reality the assembly program language was a set of synthetic computer instructions. Although these systems greatly simplified programming, the programmer was still plagued with the many details dictated by the computer.

Automatic coding languages of today are on the threshold of relieving the programmer of these details. The structure of these new languages are very much like English. By using a combination of English words and phrases to form sentences, the programmer now needs only to "describe" a procédure for the computer to follow. This procedure together with a description of the data is then given to a special computer program for processing. This special program, commonly called a compiler, translate the English problem description and generates a program of computer instructions.

Such a compiler is provided for the GE-225. Its General Compiler evolved from two noteworthy language efforts - the Common Business Oriented Language (COBOL) and the Algorithmic Language (ALGOL). Both languages were developed by voluntary committees of computer manufacturers and users and reflect the recent trend toward "common" compiler languages.

The language presently available with the General Compiler is based primarily on COBOL, since COBOL satisfied the needs of a broad spectrum of data processing applications. To accommodate the demands of more technical

applications, Boolean expressions, floating point arithmetic, and the ability to express equations were incorporated into the format of COBOL. Therefore, one may say that the present version of the General Compiler can accept programs written in one, two, or in a combination of two language forms.

Those programmers familiar with COBOL recognize that it is well suited for creating and processing data files. ALGOL, on the other hand, provides an excellent means for expressing complex mathematical relationships. Recent investigations by the Integrated Systems Project of General Electric's Manufacturing Services uncovered an area of applications which require neither extensive data file processing nor profound mathematics but rather an unwieldy number of sequential decisions.

To cope effectively with these decisions the ISP team devised a tabular language. The purpose of this language was to depict, by means of tables, the relationships of logical decisions. The new language was appropriately termed TABSOL for Tabular Systems Oriented Language. Since its creation TABSOL has been used by many departments of General Electric to analyze and solve problems in product engineering, manufacturing methods, cost accounting, and production control. The application of decision tables is continually growing. Recent studies show that they provide a concise method for supporting the logic of other data processing applications. For example, decision tables may be used to specify the transfer of control associated with the values of one or more fields, to control the printing of detail and summary lines of a report, or to interrogate the sort keys in a multi-file system. At

the Computer Department we have found decision tables a valuable tool in designing and implementing the General Compiler.

Decision tables represent a third language for the General Compiler. They may be used by themselves or in conjunction with the features of the compiler language. The specifications outlined in this manual pertain mainly to the table entries and imply and require a knowledge of the General Compiler. Therefore, this manual should be used as a supplement to the GE-225 General Compiler Manual, CPB-123 (5.5M10-60).

II. DECISION TABLE FORMAT

The format of a decision table is given in Fig. 1. In concept a table is an array of blocks divided into four quadrants by a pair of double lines. The vertical double line separates the decisions or "conditions" on the left from the "actions" on the right. The horizontal double line isolates variables from associated operands which will appear in the blocks and rows below. A condition then is a relation between a variable appearing in a primary block and an operand appearing in a corresponding secondary block. For example, we may write AGE in primary block 1 and EQ 26 in secondary block 1. In doing this, we are stating a condition. Verbally, we are asking "if age equals 26". An action, on the other hand, is a statement of what is to be done. By writing AGE in a primary action block and 26 in its associated secondary block, we are stating that "the value 26 is to be assigned to age".

It is interesting to note, at this point, the English interpretation given to the vertical lines. The left-most line may be thought of as representing the word IF. Those lines to the left of the vertical double line may be taken to mean AND; the vertical double line itself the word THEN. Since actions are sequential entities, the lines separating them may be interpreted as semicolons and the right-most line, which actually terminates the actions, as a period. With this in mind, each secondary row becomes an English sentence. For example, each row now reads:

> "IF condition-1 is satisfied AND condition-2 is satisfied AND . . . AND condition-k is satisfied THEN perform action-1; action-2; . . .; action m."

If any condition within a row is not satisfied, the next row is evaluated

DECISION TABLE FORMAT

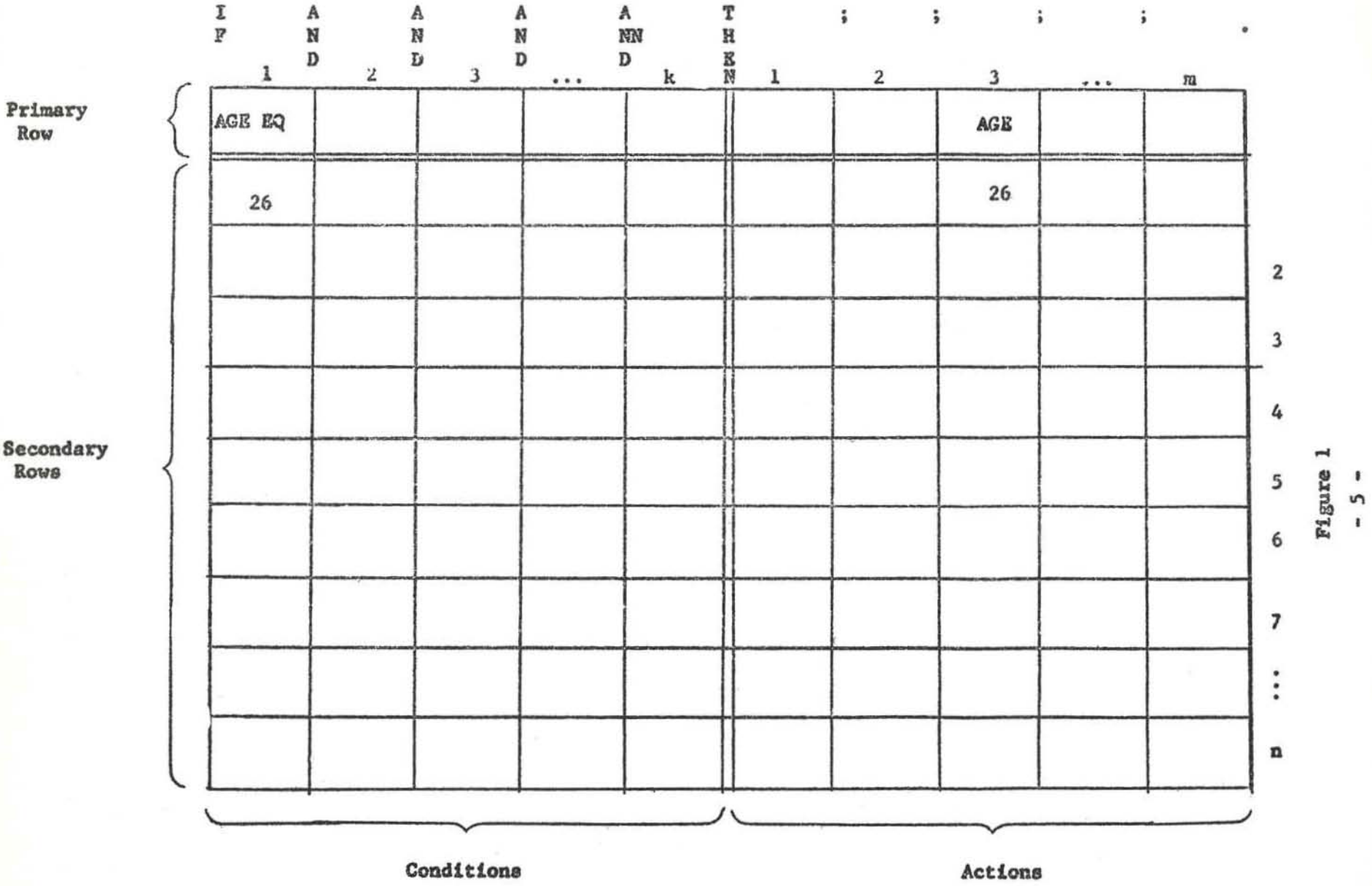

Figure 1

and so on until all the rows are depleted. When this happens the table is said to have "no solution". The table is considered "solved" when all the conditions of a row are satisfied and their associated actions performed.

Before considering the conventions used to formulate conditions and actions, an example may help develop insight into the nature of decision tables and the manner in which they may be used with the General Compiler. In this example (Fig.2) we are searching a master employee file (recorded on magnetic tape) to determine the number of male employees who fall into the following job categories.

Job Level	Years Experience		Title
6		2	Programmer
7		3	Programmer or Analyst
8	More than	3	Analyst
9	More than	4	Analyst or Manager
10	More than	4	Manager

For each employee we find having any of these qualifications, we are to write his department number, name, title, level, and experience on the computer's typewriter. At the end of the run the totals for each of the categories are to be also put on the typewriter.

The core of this problem is the decisions that must be made on the information stored in the records of the master file. These decisions are conveniently expressed above in narrative form. With only minor alteration this form becomes the program statement of our problem. The table and sentences are punched into 80-column cards exactly as they appear in Fig.2. When this is done they may be given directly to the compiler for processing.

As illustrated in our example, General Compiler sentences may be used to support the logic of the table. These sentences accomplish the following:

GENERAL ELECTRIC

GENERAL COMPILER
SENTENCE FORM

PROGRAM: Sample Decision Table
DATE:
PROGRAMMER:
COMPUTER:
PAGE: OF

```
SEQUENCE
NUMBER
     5  PROCEDURE DIVISION.
    10      OPEN INPUT MASTER-FILE.
    15  GET-RECORD. READ MASTER-FILE RECORD IF END FILE GO TO END-RUN.
    20      IF FEMALE GO TO GET-RECORD.
    25      EXPERIENCE = 60 - YR-EMPLOYED + PREV-EXP.

    30  TABLE EXAMPLE. 3 CONDITIONS 2 ACTIONS 5 ROWS.

    35      LEVEL EQ EXPERIENCE                          I  GO TO
    40      6        EQ 2    PROGRAMMER                  1  TYPE-OUT
    45      7        EQ 3    PROGRAMMER OR ANALYST       2  "
    50      8        GR 3    ANALYST                     3  "
    55      9        GR 4    ANALYST OR MANAGER          4  "
    60      10       GR 4    MANAGER                     5  "

    65      GO TO GET-RECORD.
    70  TYPE-OUT. WRITE DEPARTMENT NAME TITLE LEVEL EXPER. ENCE ON TYPEWRITER.
    75      TOTAL(I) = TOTAL(I) + 1.
    80      GO TO GET-RECORD.
    85  END-RUN. CLOSE MASTER-FILE.
    90      WRITE TOTAL(1) TOTAL(2) TOTAL(3) TOTAL(4) TOTAL(5) ON TYPEWRITER.
    95      STOP "END RUN".
```

Figure 2

Computer Department, Phoenix, Arizona CA-13 (10-60)

OPEN --- Declares that the MASTER-FILE is input and since the file is recorded on magnetic tape, validates the tape labels.

READ --- Delivers the next record from the MASTER-FILE and tests for an end-of-file sentinel. When this sentinel is detected, sequential program execution is interupted and control passes to the portion of the program labeled END-RUN.

IF --- Eliminates those data records which contain information about female employees. The word FEMALE (also PROGRAMMER, ANALYST, and MANAGER used in the table) represents a special kind of condition and will be explained later in the manual.

EXPERIENCE = --- Calculates the employees total experience and assigns the value to the field named EXPERIENCE.

The word TABLE informs the compiler that it must process a decision table; EXAMPLE is a name or label which was given to the table. The size of the table is stated next by giving the number of conditions, actions, and rows contained in the table. This information is used only by the compiler and is not executed by the compiled program.

Table execution begins at row 1 (sequence number 40). Using our narrative definition of a table, row 1 is interpreted as follows:

> "IF the job LEVEL field equals (EQ) 6 AND the EXPERIENCE field equals (EQ) 2 years AND the employee's title is PROGRAMMER THEN assign the value 1 to the subscript I; GO TO the part of the program having the label TYPE-OUT."

If one of these condtions cannot be satisfied, row 2 is evaluated starting again with the left-most condition. Sequential execution of the rows continues until either all conditions in a given row are satisfied or

all rows are exhausted. When the latter situation occurs, the sentence immediately following the table is executed. Proceeding from here the sentences in our example accomplish the following:

GO --- Interrupts sequential program execution and passes control to the part of the program labeled GET~RECORD.

WRITE--- Writes the current contents of the DEPARTMENT, NAME, TITLE, LEVEL, and EXPERIENCE fields on the computer's typewriter.

CLOSE--- Rewinds the MASTER~FILE and performs the file's closing conventions.

STOP --- Terminates processing and writes the words END RUN on the typewriter.

By General Compiler standards this example represents relatively simple conditions and actions. In formulating these entries, the programmer may take full advantage of the compiler's capabilities. The remaining sections of this manual are devoted to defining the conventions and manner in which conditions and actions may be formed and entered in tables.

III. BASIC CONCEPTS

Since decision tables are used in conjunction with the General Compiler language, we must first look at the foundations of this language before considering the counterparts that may appear in a table. The compiler's language, like most natural languages, is a body of words and a set of conventions for combining these words to express meanings. Its structure or "syntax" closely resembles the rules of English grammar, and its body of words may be appropriately termed a "vocabulary". The purpose of this section is to show how words are formed and how they may be used to express a desired meaning.

Characters

The basic units of our language are the characters used to form words and symbols. The character set includes the letters of the alphabet (A, B, C, ... Z), the numerals (0, 1, 2, ..., 9), and the special characters shown in Fig. 3. Special characters are presented in more detail as they are encountered in the manual.

Words

The words of a typical General Compiler program fall into one of two categories: the vocabulary of the compiler and the vocabulary used by the programmer. The programmer's vocabulary will consist mostly of arbitrary names given to his data and sections of his program. The compiler's vocabulary, on the other hand, is predetermined and explicitly defined in this manual. Since the compiler, by nature of its designers, is a mistrusting mechanism, the programmer must define the words he uses too. This is done, not by writing a manual, but instead by merely

SPECIAL CHARACTERS

Character	Meaning	Card Code
Δ	Space or blank	Space
.	Period - Decimal point	12-3-8
,	Comma	0-3-8
"	Quotation Mark	3-8
~	Hyphen	5-8
(	Left Parenthesis	0-5-8
)	Right Parenthesis	0-6-8
+	Addition	12
-	Subtraction - Minus Sign	11
*	Multiplication	11-4-8
/	Division	0-1
=	Assignment	6-8
|	Vertical Table Line	12-4-8

Figure 3

filling out a data description form. Once these "data names" are defined, they may be filed either on 80-column punched cards or on magnetic tape and used over and over again. The data description file then is a "dictionary" since it contains the definitions of the words used by the programmer. Furthermore, this dictionary may be revised without redefining all of its entries. This is accomplished by a special service routine which accepts corrections, insertions, and deletions as long as they are written on the compiler's data description form.

Our two categories of words may be illustrated by the following sentence taken from the program example given in Fig. 2.

GET^RECORD. READ MASTER^FILE RECORD IF END FILE GO TO END^RUN.

Here, the words READ, RECORD, IF, END, FILE, GO, and TO belong to the vocabulary of the compiler; whereas, the words GET^RECORD, MASTER^FILE, and END^RUN belong to the programmer's vocabulary. The compiler will assume that MASTER^FILE is a data name due to the word's position in the sentence. It will then search the data description to verify its assumption and to determine the characteristics depicted by this word. Not finding a match in the data description results in an error message typed on the computer's typewriter. The words GET^RECORD and END^RUN will be interpreted as sentence-names due to their position in the program. Once again, the compiler will attempt to verify its findings by checking each transfer to make certain that they lead to properly defined sentence names. The consequence of an undefined sentence name is likewise an error message on the computer's typewriter. The compatability checks mentioned here are only two of many which the compiler performs to insure unquestionable results in the programs which creates.

Formation of Names

As previously mentioned, data names are words representing data (files, records, fields, elements, constants, arrays of values, etc.) and are arbitrarily assigned by the programmer. They are formed from the following characters.

Letters	A, B, C, ..., Z
Numerals	0, 1, 2, ..., 9
Hyphen	-

To avoid error messages and possible re-compilation, the programmer should choose data names that

1. Do not exceed 12 characters,
2. Do contain at least one letter,
3. Do not begin or end with a hyphen.

To insure a properly defined program, all data names should be recorded and their characteristic data described on the compiler's data description form. The programmer also should be careful not to use the compiler's vocabulary as data names.

In addition to data names, the programmer is free to name sentences, tables, and other "procedures" in his program. With one exception these names are formed like data names. Since procedure names are judged from their position in the program, they may be formed from only the numerals, 0 through 9.

Constants

The values associated with data names generally change during the actual running of a compiled program. It is for this reason that they

are sometimes called "variables". A constant, as opposed to a variable, is a specific value and does not change within the scope of a program. Constants may be one of two kinds: a literal, or a named constant.

A literal is a value itself rather than a name given to a value. Literals may be numerical, alphabetic, or alphanumeric - i.e., composed from the character set of the computer. All non-numeric literals should be enclosed in quotation marks (") to avoid having the compiler confuse them with data names. The conventions for forming literals are the following:

1. Non-numeric literals are limited to 30 characters, excluding the quotation marks.
2. A numeric literal not enclosed in quotation marks is assumed to be a number. Numbers may contain not more than one decimal point and a minus sign. Unsigned numbers are considered positive. Excluding decimal points and minus signs, numbers must not exceed 11 decimal digits.
3. Numbers may be treated as floating point by writing them as a power of ten - i.e., a number or decimal fraction followed by a power of ten exponent. For example, the number 230100 might be written as 2.301E5 which is equivalent to 2.301 multiplied by 10^5. The exponent part, indicated by the letter E, may contain a minus sign to show a negative exponent. The value range of an exponent is limited to $\pm$ 75. Excluding the decimal point, the minus sign, and

the letter E, the fractional part of a power of ten number must not exceed nine decimal digits. To distinguish data names from floating point numbers, data names should not be formed from only the numerals and the letter E.

4. An alphanumeric literal may not contain an embedded quotation mark since the enclosing quotation marks are used to determine the size and content of the literal.

A named constant is a constant which has been given a name. Named constants are defined by means of the data description and may include any character belonging to the character set of the computer, including the quotation mark. Like literals named constants may be numeric, alphabetic, or alphanumeric. They are unlike literals in that they may be any length.

Subscripts

Subscripts provide a convenient method to reference individual values contained in a list or in an array of values. The variable, I, employed in the decision table of Fig. 2 is a subscript used just for this purpose. Since five totals are to be accumulated, one name was assigned to all five, namely, the data name TOTAL. Whenever reference was made to a particular total, the data name TOTAL was followed by the subscript I. This is illustrated in the expression

TOTAL (I) = TOTAL (I) + 1.

and the sentence which prints all five totals on the typewriter. From this example, it follows that subscripts, like data, may be given names. In fact the same rules that govern forming data names apply to naming subscripts.

Since subscripting is a positional notation, the range of any subscript is limited to the values 1, 2, 3, . . ., n (where n is the maximum number of values in a list). This does not mean that subscripts are limited only to integers. If a subscript is not defined as integer by means of the data division, the compiler will automatically provide coding to truncate its value to an integer. Furthermore, subscripts are not restricted to a single variable name. Arithmetic expressions may also be used as subscript. For example,

```
RATE (P+1)
K ((X-3)*P**3)
A (J)
```

are legitimate forms of subscripts.

Up until now, only one-dimensional subscripting was considered. Values in multi-dimensioned arrays may also be referenced by subscripts. For example, an array in which values are ordered

A_{11} A_{12} A_{13} A_{14} A_{15}

A_{21} A_{22} A_{23} A_{24} A_{25}

A_{31} A_{32} A_{33} A_{34} A_{35}

A_{41} A_{42} A_{43} A_{44} A_{45}

A_{51} A_{52} A_{53} A_{54} A_{55}

might be subscripted as A (J,K), where K is the columnar subscript and J the row. To refer to value A_{35}, J would have to equal 3 and K equal 5.

Preceeding examples show that subscripts are enclosed in parenthesis and separated by commas. This notation permits the compiler to distinguish subscripts from other elements in the language.

Truth-Values

There is a class of variables which, through either usage or definition, may assume only the numerals 1 or 0. The value 1 is said to be their true state and the alue 0 their false state. The words END FILE of the READ sentence in Fig. 2 is such a variable. When the OPEN sentence is executed, END FILE is set to its false state and remains so set until the end-file condition is encountered. At this time, it is set to its true state.

Variables having truth-values are termed "True-False" variables. END FILE is a convenience provided by the compiler; the programmer may also formulate his own true-false variables by merely listing them under the heading TRUE^FALSE in the data division. They may be named according to the rules given for data names.

Arithmetic Expressions

Arithmetic expressions are rules for computing numerical values. They are formed from variables, numbers, functions, and symbols representing addition, subtraction, multiplication, division, and exponentiation. For example, in the expression

PREM^HRS * 2.50 + OT^HRS * 3.75

PREM^HRS and OT^HRS are variables; 2.50 and 3.75 numbers; and + and * symbols for addition and multiplication. If PREM^HRS were 40 and OT^HRS were 4, the expression becomes 40 * 2.50 + 4 * 3.75 and after performing the arithmetic, reduces to the value 115.00. To save this value, a programmer might write

GROSS^PAY = PREM^HRS * 2.50 + OT^HRS * 3.75.

The presence of the = symbol tells the compiler to assign 115.00 to the variable GROSS^PAY. When expressions are written in this form, they are called "assignment statements".

The arithmetic permitted in an expression is stated by the following symbols:

Symbol	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

In addition to arithmetic, the following mathematical functions may be used.

Symbol	Function
SIN	Sine
COS	Cosine
ATAN	Arctangent
SQRT	Square Root
EXP	Exponential
LOG	Common Logarithm
LN	Natural Logarithm
ABS	Absolute Value

Arithmetic expressions are evaluated from left to right according to the following priority:

1. Exponentiation and Functions
2. Multiplication and Division
3. Addition and Subtraction

Parentheses may be used to establish a precedence other than the one above. When they are used, the evaluation is performed from the innermost to the outermost pair but still from left to right within a given pair.

Relational Expressions

A relational expression is a statement of magnitude between two values. For example, FICA GR 144.00 is a comparision between the variable FICA and 144.00. The symbol GR stands for the relation "greater than". Other relations may be stipulated by

Symbol	Relation
EQ	Equal to
GR	Greater than
LS	Less than
NEQ	Not equal to
NGR	Not greater than
NLS	Not less than

To have meaning, relational expressions must be stated as conditions. The expression FICA GR 144.00 tells us nothing. However, when it is written as

IF FICA GR 144.00, GO TO ADJUST-PAY

we know immediately what is intended. By definition then, relational expressions are conditions and when evaluated always give a truth-value.

Relational expressions may be explicitly stated or implied. FICA GR 144.00 is an explicit statement of magnitude. In the program example of Fig. 2, implied relations were stated by the words FEMALE, PROGRAMER, ANALYST, and MANAGER. An implied expression is formed by giving a name to a value, a range of values, or to a series of values and ranges. Once the name and its values are defined in the data division, it may be used to mean its associated values. Implied relations are termed "condition-names" since a name was given to a condition, i.e., a value, of a variable. The

variable from which the value is taken is called a "conditional variable". Therefore, writing PROGRAMMER (fig.2) in a decision table block is the same as writing an expression which will compare the TITLE field with the value associated with the title, programmer.

Logical Expressions

Logical expressions provide a convenient method for obtaining truth-values. They are formed by combining true-false variables and relational expressions with the logical operators AND, OR, and NOT. The expression (Fig.2)

PROGRAMMER OR ANALYST

is a logical expression which is true when an employee's TITLE field indicates that he is either a programmer or an analyst.

The rules governing the evaluation of logical expressions may be expressed as follows:

p	F	F	T	T
	F	T	F	T
NOT p	T	T	F	F
p AND q	F	F	F	T
p OR q	F	T	T	T

where p and q are a combination of true-false variables, relational expressions, or logical expressions.

Logical expressions are evaluated from left to right with the logical operator AND having precedence over the OR. Parentheses may be used for grouping or establishing a precedence of evaluation other than the one mentioned previously. When they are used, the evaluation proceeds from left to right from the innermost pair to the outermost pair.

IV TABLE ENTRIES

The previous section outlined the elements of the General Compiler language and briefly showed how they might be used. In the introduction, it was mentioned that these same elements may be employed within the blocks of decision tables. The purpose of this section is to show how this may be done.

Formation of Conditions

By definition, a condition is a relation between a primary block entry and some corresponding secondary block entry. A condition, like a relational expression, may be either true or false. True conditions are said to be "satisfied" and false conditions "not satisfied". From this definition, a condition may be either a relational expression, a logical expression, or a true-false variable since these are the only elements that yield a truth-value.

The formats noted below show how these expressions may be split between primary and secondary blocks to form conditions. In these examples, the word "operand" stand for either a variable (data name or subscripted data name), a constant (literal or named constant), or an arithmetic expression. The word "relation" signifies one of the relational operators - EQ, GR, LS, NEQ, NGR, or NLS. Since arithmetic expressions may be operands of relational expressions and relational expressions as operands of logical expressions, it necessarily follows that arithmetic expressions may appear in logical expressions.

Format	Example
Operand-1 Relation	LEVEL EQ
Operand-2	10

Unit	Example
Operand-1 Relation Operand-2	EXPERIENCE GR 4
Operand-1 Relation Operand-2 OR Operand-3	TOTAL (I) NLS PT(1) OR PT(2) or PT(3)
Operand-1 Relation-1 Operand$_2$ OR Relation-2 Operand-3 ...	(X+Y) ** 3 GR P+1 OR LS Q(I)
No Entry Condition-name	 PROGRAMMER
NOT Condition-name	NOT FEMALE
No Entry True-False Variable	 REQ-1
NOT True-False Variable	NOT END INVENTORY FILE
No Entry Logical Expression	 PROGRAMMER OR ANALYST
NOT Logical Expression	NOT X GR Y OR X LS (Z+1)

Formation of Actions

Actions are statements of the things to be done when all the conditions of a row are satisfied. The scope of an action may be one of three kinds: implied assignment, procedural, or input-output. The only action presented so far was assignment. The other two are extensions of General Compiler sentences and will be mentioned here only briefly. The compiler manual should be consulted for a more detailed presentation.

1. Value Assignment. Value assignment is an implied function between associated, primary and secondary block entries. By placing a data name in a primary block and some number in a secondary block, for example, I and 1 of Fig. 2, the compiler automatically produces coding to assign the numer to the data name. In the case of our example, 1 is assigned to the subscript I. Other examples of value assignment are given below. In these formats the word variable implies either a data name or a subscripted data name and the word constant either a literal or a named constant.

Format	Example
Variable / Constant	I / 1
Constant / Variable	"COPPER" / MATERIAL
Variable / Arithmetic Expression	ALPHA (I,J,K) / SIN THETA + (X/P)**2

Format	Example
Arithmetic Expression	PI * R**2
Variable	AREA#1
True-False Variable	SWITCH#7
Truth-Value 1 or 0	1
Truth-Value 1 or 0	0
True-False Variable	BETA#REQ

2. Procedural actions. Procedural actions provide the means for interrupting the normal execution sequence of a table. Any of the following compiler verbs may be used for this purpose.

GO TO
PERFORM
STOP

The GO verb stipulates an unconditional transfer to a specified part of the table or program. Its destination may be a sentence name, table name, or the row number of a particular table The format of the GO entry is as follows:

Format	Example
GO TO	GO TO
Sentence Name	TYPE#OUT
GO TO	GO TO
Table Name	TABLE 23
GO TO	GO TO
Row of Table	ROW 7 TABLE BETA

The other form of a procedural control is the PERFORM verb. The PERFORM specifies a transfer to some destination, the execution of a table or a set of sentences at that destination, and a return to the action block following the PERFORM. The sentences or tables acted upon are by definition a "closed procedure" - i.e., they have a single entrance point and a defined exit point. Conventions for writing closed procedures are given in the next section. Legitimate forms of the PERFORM action are

Format	Example
PERFORM Sentence Name	PERFORM GROSS-PAY
PERFORM Table Name	PERFORM ERROR TABLE

The STOP verb may also be used as an action. It may be placed in either a primary or secondary block. When it is used, no other action may appear with it in the same action column. The STOP terminates processing temporarily or permanently according to what action is taken at the computer's console.

3. Input-Output Actions. Input and output actions are compiler verbs that control the flow of data to and from the computer. They read, write, and validate tape labels of data files assigned to peripheral input-output devices. When data files are referred to from an action block, they must be defined according to the environment and data division specifications listed in the General Compiler manual. The formats of input-output actions are illustrated by the following:

Format	Example
READ File Name	READ MASTER~FILE
OPEN INPUT or OUTPUT File Name	OPEN INPUT MASTER~FILE
CLOSE File Name	CLOSE MASTER~FILE
File Name READ, CLOSE, or OPEN verbs	MASTER~FILE READ
WRITE Record Name	WRITE DETAIL~LINE
Record Name WRITE	TRANSACTION WRITE

The Skip and Repeat Operators

The skip operator makes it possible to show that a condition or action is not to take part in the evalution of a row. This is done by placing a hyphen (~) in the concerned condition or action block. The compiler then will skip this block and proceed to the next.

The repeat operator is a shorthand method to indicate that a condition or action in the block above is repeated. This is shown by entering a ditto mark (") in the block below the one that is to be repeated. This notation was used with the GO TO action in the sample table of Fig. 2.

V THE TABLE AS A PROGRAM

Up until now, only components of tables were presented. It was learned in Section II that General Compiler sentences could be used to support the conditions and actions of tables, and the preceeding section mentioned tables as closed procedures. This sect on relates these topics to tables and tables to compiler programs.

Block Conventions for Writing Expressions

1. Words, abbreviations, and symbols of the compiler's vocabulary should not be used as names. They may be combined with other characters to form names.

2. The words in an expression should be separated by at least one space. More than one space is permitted. The space separator is optional if the words are bound by

+ - * / ** ~ ( ) " = , |

3. Subscripts should be enclosed in parentheses. They may be written adjacent to (without a space separator) or apart (with space separators) from their associated data names. Individual subscripts in a list of subscripts should be separated by commas.

4. When two arithmetic expressions appear side by side as in a series, they should be separated by commas.

5. All columns of a table should be bound by the vertical table line, | (12-4-8 punch).

6. The skip and repeat symbols, ~ and ", should be the only entry , other than spaces, in a block.

Conventions for Placing a Table in a Program

1. Tables are written on the General Compiler Sentence Form.
2. A table is preceeded by the word TABLE. Naming tables is optional. When a table is given a name, the name may preceed or follow the word TABLE. The word

   TABLE,
   name TABLE, or
   TABLE name

   should be followed by a period.
3. The table's size is given next and should be placed on the same line as the table's name. The size may be written in one of two ways:

   kkk CONDITIONS mmm ACTIONS nnn ROWS.

   or

   (kkk, mmm, nnn).

   Both forms are terminated by a period. The order of writing the number of conditions, actions, and rows is optional in the first case since each can be identified. However, order is important in the second form since the compiler interprets the first number enclosed in parentheses as the number of conditions, the second as actions, and the third as rows. Conditions, actions, and rows are numbered sequentially beginning with 1. Row 1 is the first secondary row; the primary row is not counted in the row count.
4. General Compiler sentences should not be placed between the word TABLE and the primary row of the tables.
5. The double vertical lines that separates conditions from actions may be represented by one or two 12-4-5 punches.

6. The size of each block may vary from column to column and row to row.

7. The only limit on the size of a table is row width. Since the compiler prints a listing of compilation, the recommended row width is 120 characters including card sequence number. Maximum row width is 1200 characters.

   Since the table form is an image of an 80-column punched card, a hyphen (~) is placed in column 7 of the form to show that a row is contained on more than one card. In this case, no table column may be split across cards. Each card is to contain a sequence number to insure proper card order. When rows exceed one card, the sequence number of the first card is only printed. Sequence numbers of succeeding cards are stripped out. The row is then printed as a multiple of 120 characters with an integral number of table columns per 120 characters.

8. Expressions too long or complex to be written in blocks may be written after the table's name and size and be executed from the table by means of the PERFORM verb. In addition to expressions, any General Compiler sentence may be used and executed in this manner. To indicate the start of the table the word BEGIN is to follow the list of expressions and sentences. This format may be illustrated by the following:

```
TABLE name.  kkk CONDITIONS mmm ACTIONS nnn ROWS.

... General Compiler Sentences and Expressions - May be
    executed only from the confines of the table.

BEGIN
```

DECISION TABLE

Closed Procedures

Fig. 4 outlines the format of a closed procedure. By definition a closed procedure may be acted on only by the PERFORM verb. It contains one entrance point and one exit point. In fig. 4 these are indicated by the words BEGIN and END TABLE name. BEGIN and END also act as sentence names and may be referred to from within the procedure body.

Expressions too long to be placed in the blocks of a table may be written in the procedure head and executed from the procedure body by means of the PERFORM verb. As such, they must be given names. In addition to expressions any General Compiler sentence may be written in the head and executed accordingly.

The procedure body contains the table. As shown in Fig. 4 compiler sentences may preceed and follow the table. Execution is sequential starting with the sentence or table after the word BEGIN and proceeds until the exit END TABLE is reached. It is at this point that control is reverted to the PERFORM verb which originally referenced the procedure. Any unconditional transfer from within the procedure to the outside is undefined. However, PERFORM verbs in the body may reference other closed procedures.

Closed procedures should be written apart from the main program.

DECISION TABLE AS A CLOSED PROCEDURE

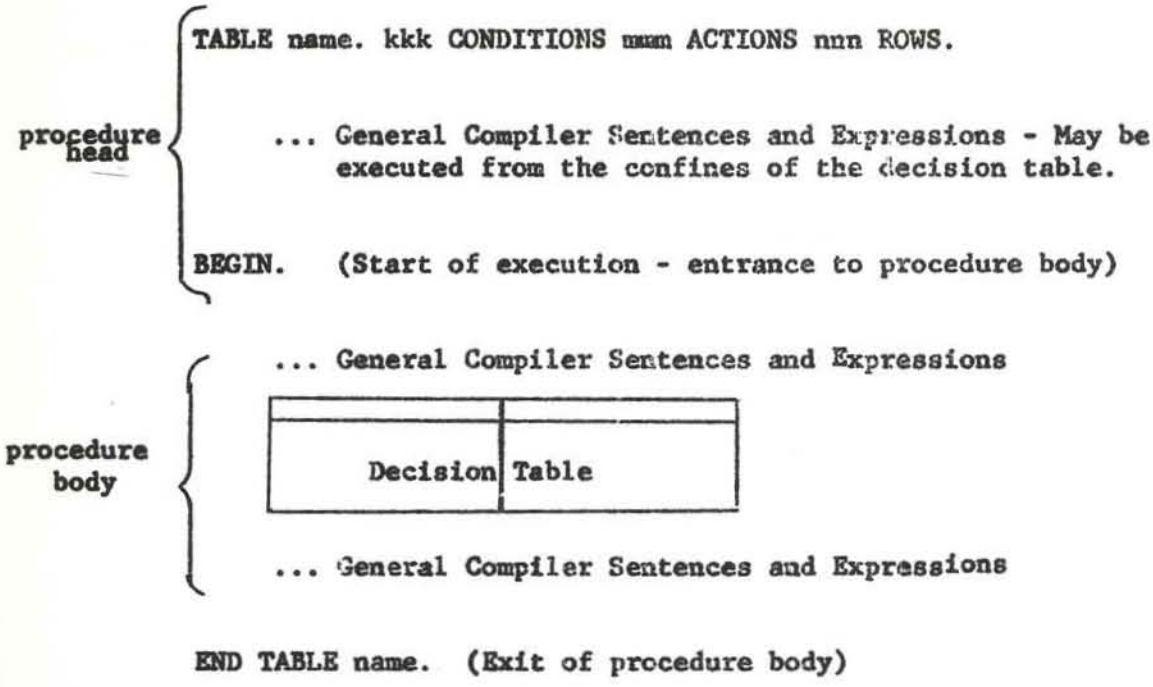

Fig. 4

IDENTIFICATION

LOGTAB COMPILER -- U00400
29 July, 1959

Jane E. King
LST-G Dept., General Electric Company
Schenectady, New York

PURPOSE

LOGTAB is a 704 program which compiles a program expressed in LOGTAB type logic tables for the IBM 704. The output is a tape containing the SAP program which can be assembled directly or which can produce cards by means of the off-line punch. The resulting program operates in conjunction with a set of standard LOGTAB subroutines. (See DF-59LS23).

RESTRICTIONS

1. Input may be on-line (Sense Switch 2 down) or off-line (Sense Switch 2 up) on tape 2.

2. Output is on tape 7.

3. The program is designed to operate within the TOP system, if desired.

4. Any number of tables may be compiled at one time.

5. If any one table being compiled results in a program longer than 1000 words on an 8192 word 704, a drum is required. The limit on a 32,768 word machine is about 6000 words before drums are required but such a table is not conceivable.

6. Only 35 columns can appear in one table. If more than 35 are needed, the table must be split and the 35th column should direct the program to the next table.

7. While operating within a table the index registers are not preserved.

8. When executing external subroutines which the table calls for in Quadrant D, index register 2 must be preserved within the subroutine. In all other cases, the index registers are free. All subroutines are entered by the compiled program with index register 4.

9. When addition or subtraction is indicated in Quadrant C, the compiled program assumes that fixed point arithmetic is to be used. If floating point is desired, the ADD instructions must be changed manually to FAD.

10. The LOGTAB subroutines which are needed for execution are not included on the output tape with the program.

11. All symbols (eg. XYZ, PQR, etc. under "Allowable Table Entries") used as elements in tables must consist only of alphabetic and numeric characters and at least one character must be alphabetic. In other words, the special characters + - (. $) , / * = may not be used in symbols.

12. The shortest and fastest operating program will result when 1) no symbolic subscripts are used, 2) the elements in Quadrant B are literal values (not symbols or subroutines), and 3) the elements in Quadrant D are symbols or subroutines (not literals). Also, increasing the number of columns in a table has very little effect compared to increasing the number of rows.

INPUT FORMAT

Four distinct types of cards are required by the compiler.

Heading card

The first card for each table must be a heading card which contains identification as follows:

Col 1 - 5 : the word TABLE

Col 7 - 11 : a symbol which is the name of the table. The first instruction of the compiled program will use it as the location symbol. If the symbol is less than five characters long, the positioning within these columns is not restricted.

Col 13 - 16 : symbol designation. For each table compiled, LOGTAB generates three symbols. It automatically assigns))))0,))))1, and))))2 for the first table, and))))3,))))4, and))))5 for the second, etc. This feature will be overruled if a number appears in these columns of the heading card. For example, if the card contains 24, then the first table will be assigned)))24,)))25, and)))26; the next table, if it has no number of its own, will be assigned)))27,)))28, and)))29. The first digit of the number to be used must be in column 13, and no non-numeric characters may be used. (See last paragraph of Output Format for usage.)

Element card

The data for the elements within the table is supplied on the element cards with only one piece of data per card. The cards for all the elements in one row must be together. In addition, the card for Quadrant A or Quadrant C must be the first card supplied for a row. The ordering of the Quadrant B or Quadrant D cards within a row is not fixed. The groups of cards for rows may be in any order except that all decision rows must be given before the first result row and the group for an OR row must immediately follow the group for the row it is OR'd with. The format of an element card is as follows:

col 1 : a letter (A, B, C, or D) to designate the quadrant in which this element appears.

col 3 - 4 : the number of the row in which this element appears. If only one digit is needed, it may be expressed as (zero, N) or (blank, N) or (N, blank).

Col. 6 - 7 : the number of the column in which this element appears. This number must be less than or equal to 35. If only one digit is needed, it may be expressed as (zero, N) or (blank, N) or (N, blank).

col 10 and following : the element itself. The element must start in col 10 and continue without spaces. For Quadrant A elements, the element and the condition of the test both appear on the card and they must be separated by at least one blank but any number of blanks may intervene.

There are several additional rules for expressing the OR condition. If the same element is to be tested against more than one value in the same column with

the same condition, only one Quadrant A card should be supplied. For example, if XYZ may be equal to 24 or equal to 36 in column one, then the following element cards should be used:

```
A 04 01   XYZ   =
B 04 01   24
B 04 01   36
```

If the same element is to be tested against more than one value in the same column with a different condition, or if a different value is to be tested, then col 10 - 12 must contain "OR," and followed by the element and/or the condition of the test without blanks. The following shows the cards required for 1) XYZ equal to 24 or XYZ less than or equal to 36; and 2) XYZ equal to 24 or PQR equal to 36.

```
A 04 01  XYZ   =          A 04 01  XYZ   =
B 04 01  24               B 04 01  24
A 05 01  OR,   (=         A 05 01  OR,PQR  =
B 05 01  36               B 05 01  36
```

If no element is given in Quadrant C (i.e., no value is to be modified as a result of the action in Quadrant D), the element card must be supplied but column 10 must be blank. If the element in Quadrant D is a dash (minus sign), it is not necessary to include an element card and the compiler ignores it if it is given.

Since a blank terminates an element and also terminates the condition of the test, the columns following the blank are not used by the compiler.

End Card

The last element card for each table must be followed by an end card which had "END" in columns 1 - 3. This card terminates the compilation for the table and initiates output of the resulting program.

Blank Card

A blank card must follow the end card for the last program to be compiled. This is the only indication available to the compiler that all of the input has been processed.

OUTPUT FORMAT

The compiled program consists of four distinct sections. The first section is the program (consisting of calling sequences to the LOGTAB subroutines) which executes the desired table. The other three sections contain data on which the subroutines operate.

The first (program) section follows a definite pattern which is dictated by the form of the input. The first three instructions initialize the subroutines to begin a table and they are:

```
XXXXX CLA T))BS
      STO T))MK
      STO I))MK
```

(XXXXX is the name of the table given on the heading card)

Following the initialization is a group of instructions for each test row. If any element in Quadrant B for that row is not a literal, then the first instructions will be used to place the current value represented by the symbol given into the list of values to be tested against. Then, if the A element was XYZ, then the following instructions would appear:

```
CLA XYZ
TSX T)),4
PZE (location of last value)+1,N,(no. of values)
ORA (location of last bit pattern)+1,1
```

If an OR condition was indicated for this row, then MZE would appear in place of PZE. The N in the tag of the PZE instruction is a code to tell what the condition of the test should be; for exapmple, a code of 3 means equal. (See allowable test conditions for list of all codes)

Following the calling sequence for the last test row is the instruction:

```
TSX F))MK,4
```

This routine takes the resultant bit pattern for all tests and removes all one-bits except the first one. Next comes a calling sequence for each action row of the table and it has the following form:

```
TSX RT)),2
PZE (location of last action)+1,0,(no. of actions)
PZE (location of last bit pattern)+1,0,-N
```

If the action called for is a replacement of XYZ by the value found in the D Quadrant, then the calling sequence would be followed by:

```
STO XYZ
```

The N in the decrement of the third word of the calling sequence is the number of words to skip if no action is inferred in the D Quadrant. For example, in the case of the replacement of XYZ shown above, the "STO XYZ" would have to be skipped and the decrement would contain "-1".

The last instruction compiled in section one is:

```
TRA ERROR
```

This causes a transfer to an error routine if no exit was found within the table. No error routine is given in LOBTAB -- it must be supplied by the programmer if such a routine is desired.

The second section of the compiled program is a list of the test values implied in Quadrant B with one entry for each different element in a row. The second half of this section contains instructions with which to obtain the values of Quadrant D elements, one entry for each different element in a row.

The third section contains the bit patterns which correspond to the words in the second section.

The fourth section is an auxiliary portion to augment the second section when one word per entry is not sufficient. The word for a Quadrant D entry in the second section must obtain the value to be used and this is sufficient in the case of a symbol without a symbolic subscript. In this case, the entry may be of the form CLA XYZ or TSX PQR,4. But if the value is a literal (eg. 24), then the 24 would have to appear in the fourth section and the word in the second section would be CLA (location of 24 in fourth section).

Each section has an associated symbol appearing with the first word in the section. They are: first = table name, second =))))0, third =))))1, and fourth =))))2. These may be modified by the rules for generated symbols given on the heading card.

ALLOWABLE TABLE ENTRIES

The following chart shows the kinds of entries which are allowed, what they mean to the compiler, and in which quadrants they may appear:

	Entry	Quadrants	Meaning
1.	(literals)	B,D	An actual value. If no decimal point or exponent appears, it is assumed to be a fixed point number with the binary point at the extreme right. If a decimal point or an exponent appears, it is used as a floating point number. An exponent is indicated by a sign and the number of decimal places to be shifted (eg. 14×10^5 would be expressed as 14+6). Plus and minus signs may be used in a literal but the plus sign is optional.
2.	XYZ	A,B,C,D	The quantity in the cell with the location symbol XYZ.
3.	XYZ(3)	A,B,C,D	The third element in an array whose first cell has the location symbol XYZ. The compiled instruction would have the address XYZ+2. Note that XYZ and XYZ(1) are synonomous.
4.	XYZ(N)	A,B,C,D	The Nth element in an array XYZ. The current contents of the address portion of cell N is added to the location XYZ-1 to compute the address of the value to be used.
5.	XYZ(N+2)	A,B,C,D	The same as 4 above except that the address is 2 higher. The sign of the numeric portion of the subscript may be minus.
6.	*PQR	A,B,D	The result remaining in the accumulator upon exit from the subroutine entered by TSX PQR,4.
7.	-(type 2-6)	A,B,D	The value expressed by any of the forms 2-6 with the sign changed.
8.	/(type 2-6)/	A,B,D	The absolute value of a value expressed by any of the forms 2-6.
9.	-/(type 2-6)/	A,B,D	The value expressed by any of the forms 2-6 with a minus sign appended.
10.	**PQR	D	PQR is the location symbol of the cell to which the program transfers unconditionally.
11.	)XXXXXX(	B,D	A BCD literal. The 6 characters following the) will be used as a BCD constant of one word.
12.	-	B	No test is made. This implies that any value will satisfy the condition. CAUTION: Extreme care must be exercised in using the - in an OR row!
13.	-	D	No action is to be taken.

14.	+(type 2-5)	C	Increase the value expressed by any of the forms 2-5 by the value indicated in Quadrant D.
15.	-(type 2-5)	C	Decrease the value expressed by any of the forms 2-5 by the value indicated in Quadrant D.
16.	(blank)	C	No value is to be modified by the value found in Quadrant D. This implies that D contains either a **FQR or *ABC where the subroutine ABC stores its own results.

ALLOWABLE TEST CONDITIONS

The following chart shows the possible test conditions in Quadrant A, how to express them, and their codes in the output calling sequence.

CONDITION	ON INPUT CARDS	CODE
Equal	=	3
Greater than	)	2
Greater than or equal to	)= =)	1
Less than	(	5
Less than or equal to	(= =(	4
Not equal	())(	6

Each Quadrant A element card must contain a test condition and it may also indicate an OR condition. See Element Card description for the card format.

USAGE - COMPILER

If LOGTAB is being used to compile within the TOP system, a comment card must be inserted behind the program call card requesting the operator to set a tape to No. 7 and to remove it upon completion of the run. This tape will contain the output from the compiler.

The data if read on line must be placed directly behind the last card of the binary program deck.

USAGE - PROGRAM

The resulting program in SAP card format on the output tape is not a complete program. It consists only of that portion of the program which was expressed in logic tables. In addition, the compiled program must have the LOGTAB execution subroutines added to it. These subroutines contain a total of 141 words. They are not included on the output tape because several LOGTAB compilations could be made for the various parts of one complete program and this would produce duplicate copies of the subroutines.

There are three possible ways to use the output tape to get an assembled program.

1 - Use the off-line punch to produce decimal cards which can be added to the decimal deck of the remainder of the program before it is assembled.

2 - Use an off-line punch simulator on the 704 (eg. UA TCH1) to produce cards and use as in the case above.

3 - Combine the information on the output tape with the remainder of the program which is also on tape and assemble from the

resultant tape. The LOGTAB compiler does not rewind tape 7 when output is initiated and this allows one to have the remainder of the program on tape 7. This tape could then be positioned at the end of the remainder of the program and the LOGTAB-compiled program could be added to it. The compiler writes an end-of-file on tape 7 after all tables have been compiled. Tape 7 is not rewound by the compiler.

ERROR EXITS

The compiler checks for several types of errors, but regardless of the success of compilation, it calls TOP. If it is not operating in the TOP system it stops at location 4246 (octal).

If any errors have been detected, the contents of index registers 1 and 2 will indicate the type of error. The following chart shows what the possible error exits are (all in octal).

IR NO. 1	IR NO. 2	REASON
16403	-	end-of-file in reading cards if AC is positive, or failed twice in reading off line input record if Ac is negative.
(16275 (16267	- -	non-numeric codes in row or column number.
15446	-	table too large to be compiled - drum has been exhausted.
15374	-	illegal quadrant on input card.
15350	-	illegal test condition on a quadrant A element card.
13710	-	machine error in computing drum address.
13524	17023	error exit from table which interprets element on element card.
13524	6	all of input card has been read without finding a necessary blank or test condition.
13524	15104	illegal data for quadrant A. The last data card read was for quadrant A but this stop occurs because of the data on the previous quadrant A element card.
13524	14711	illegal data for quadrant C. The last data card read was for quadrant C, but this stop occurs because of the data on the previous quadrant C element card.
13524	14447	illegal data for quadrant B on last element card (read.
13524	14103	illegal data for quadrant D on last element card (read.

NOTE These index register values are for an 8192-word machine. Add 60000 (octal) for a 32,768-word 704.

TIMING

A rough estimate of the time required for a compilation can be found by computing the time required to read the input cards on-line and multiplying this time by two.

NOTE: Restriction No. 11 concerning operating time for the compiled program.