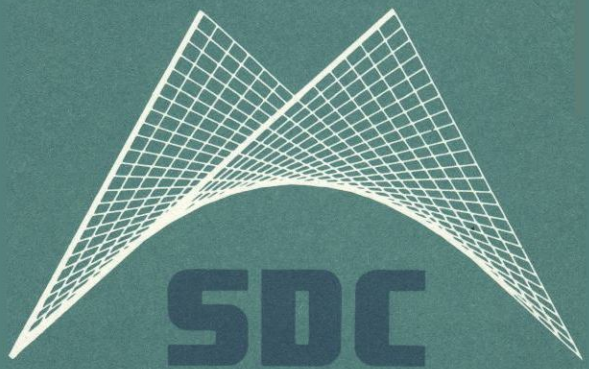


TM-555/003/00

The JOVIAL Manual

Part 3

The JOVIAL Primer

26 December 1961

TM-555/003/00

TECHNICAL MEMORANDUM

(TM Series)

The JOVIAL Manual

Part 3

The JOVIAL Primer

C. J. Shaw

26 December 1961

SYSTEM
DEVELOPMENT
CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA

(Page 2 blank)

PREFACE

Part 3, the JOVIAL Primer, is a complete and self-contained text on JOVIAL programming, with many examples and exercises. The treatment is such that an experienced programmer should have little trouble in comprehending the subject matter, although a beginner might require some additional tutoring. Answers to the exercises in the Primer will be found in Part 4, Supplement to the JOVIAL Manual.

Although the Primer contains all of the information found in Part 2, the Grammar and Lexicon, it is not meant to supersede that document, since the organization of material in the two documents differs considerably. The Primer is organized so that, in general, simple topics are presented first, more difficult (and less useful) topics later. The Grammar, on the other hand, obeys the heirarchical structure of the language and is organized for reference.

As an aid to legibility in the examples, lower case as well as upper case letters will be used though, of course, JOVIAL itself contains no such distinction. Lower case letters will be used within comments and to expand the abbreviations that the language employs.

TABLE OF CONTENTS

(Page 6 blank)

INTRODUCTION

The circuitry of a digital computer allows it to process (input, store, manipulate, and output) information under the direction of a program of instructions supplied by the programmer. Information so processed is composed of basic units called values. Because a digital computer uses the on/off states of its hardware devices in processing information, such computers represent values with binary symbols: numeric values with symbols from the binary number system; other values with more or less arbitrary codes, since no standard binary encoding schemes exist for even the most common classes of non-numeric symbols.

Digital computers thus process not values but binary symbols representing values. And since a language of binary symbols is convenient only for machines, programmers have developed more intelligible programming languages, composed of alphanumeric symbols, which can usually be automatically translated into machine language by the computer itself. These languages, closely resembling machine languages at first, have evolved in the last few years to become less machine-oriented, more procedure-oriented, and more powerful.

Such a procedure-oriented language is JOVIAL, designed by SDC for programming large, computer based command/control systems. The JOVIAL programmer thus describes and names the data his program is to process and describes the processing with familiar symbols comprising lists of declarations and statements. A JOVIAL compiler reads the declarations, creates a dictionary of data names, decides the storage allocation for the machine language symbols representing the data, and scans the statements, replacing them with sets of equivalent, machine-like operations which it then uses to produce the machine language program.

NOTATION

The grammar used to describe JOVIAL syntax consists of rules with the form:

element Φ string-of-elements

where an element either denotes or exhibits a JOVIAL form. (1) The metasymbol Φ signifies syntactic equivalence, while the colon : signifies concatenation and the semicolon ; signifies selection between adjacent elements. (2) A subscript for an element is a semantic cue, with no syntactic effect. (3) The brackets $\lceil$ and $\rceil$ group a string of elements into a single element, while the brackets $\lfloor$ and $\rfloor$ group a string of elements into a single, optional element. (4) The suffix s signifies a string of one or more of the elements to which it is appended,

while an element superscripting it is the separator, if any. (5) A space signifies a string of JOVIAL blanks, and since JOVIAL symbols are normally separated by blanks, this separation convention will not be explicitly indicated.

JOVIAL is a programming language for professional programmers, who are notoriously averse to redundant coding. JOVIAL consequently uses certain abbreviations: S for Signed; F for Floating; B for Boolean; and so on. Indeed, these abbreviations are the normal forms of the language and must be explicitly defined away if the expanded versions are to be used. For example:

```
DEFINE Boolean ''B'' $
```

To simplify the discussion, therefore, it is assumed that definitions like the above have been given. The normal, JOVIAL abbreviation is the capitalized, first letter.

ALPHABET AND VOCABULARY

JOVIAL's symbols are formed from an alphabet of 48 signs consisting of 26 letters, 1Ø numerals, and a dozen miscellaneous marks including the blank, the prime, and the dollar sign.

```
letter  ≠  A;B;C;D;E;F;G;H;I;J;K;L;M;N;O;P;Q;R;S;T;U;V;W;X;Y;Z

numeral ≠  Ø;1;2;3;4;5;6;7;8;9

sign    ≠  letter;numeral;blank;);(;+;-;*;/;.;,;';=;$
```

Certain strings of these signs are JOVIAL symbols: delimiters; identifiers; and constants, which are themselves strung together to form the clauses and sentences of the language. For legibility of layout, symbols are separated by an arbitrary number of blanks, and may therefore contain no embedded blanks. In JOVIAL, line endings perform no function so that, where necessary, a symbol may extend past the end of a line.

DELIMITERS

Delimiters (so-called because one of their functions is to syntactically delimit identifiers and constants) are the verbs and the punctuation of JOVIAL. They have fixed meanings best described in later context.

arithmetic-operator ≠ +;-;*;/;**

relational-operator ≠ EQ;GR;GQ;LQ;LS;NQ

logical-operator ≠ AND;OR;NOT

sequential-operator ≠ IF;GOTO;FOR;TEST;CLOSE;RETURN;STOP;IFEITHer;ORIF

file-operator ≠ OPEN;SHUT;INPUT;OUTPUT

functional-modifier ≠ BIT;BYTE;MANT;CHAR;SIGN;NENT;NWDSEN;ALL;ENTRY;
POSition

separator ≠ .;,;=;==;';...;$

bracket ≠ (;);(/;/);($;$);'';'';BEGIN;END;DIRECT;JOVIAL;START;TERM

declarator ≠ ITEM;MODE;ARRAY;TABLE;STRING;OVERLAY;DEFINE;SWITCH;
PROCedure; FILE

descriptor ≠ Floating;fixed;Dual;Signed;Unsigned;Rounded;Hollerith;
Transmission;Status;Boolean;Variable;Rigid;Preset;Like;Parallel;Serial;
Dense;Medium;No;Binary

IDENTIFIERS

The totality of information needed for the correct functioning of any program is called its environment. Since information in a digital computer consists of both data and instructions, it is not surprising that a program's environment consists of the data it processes, and the program itself (including any necessary super or sub-ordinate programs.) The environment of a JOVIAL program thus includes: statements, switches, and procedures, which comprise programs; and items, tables, and files, which comprise data.

The elements of the environment of a JOVIAL program are briefly described, pending more comprehensive discussion, as follows:

STATEMENTS are the instructions of JOVIAL. Each simple statement describes a complete and individual computation affecting either a designated value or the sequence of statement executions (or both).

SWITCHES are devices for describing the computation of a statement name.

PROCEDURES are bounded sets of statements performing a computational task which may be called for at any point in the program.

ITEMS are the basic units of data in JOVIAL, since each corresponds to an individual data value. An item consists of: a symbol, representing the value; a name, identifying the value; and possibly an index, distinguishing the value from similar values with the same name. Items may be organized into tables, strings, or arrays.

TABLES are matrices of item-values. The rows of a table are called entries, and an entry consists of a related set of different items. Typically, entry K ($item1_K$, $item2_K$, $item3_K$, ... , $itemN_K$) would consist of values measuring the N pertinent attributes of "object" K. All the entries of a table have the same structure in the sense that each consists of a similarly named and ordered set of items. A particular value in a table is designated by item name and entry index.

FILES are collections of information contained in 'external' storage devices, and thus provide the computer with its input and output as well as auxilliary storage. (In practice, any storage device containing information that cannot be accessed by a single machine instruction may be considered an external storage device.) In structure, a file is a linear, ordered set of records each of which is a single, composite symbol representing the (many) values of, perhaps, an entire "block" of items. The information in a file is input and output a record at a time.

Any program must, by means of symbols, refer constantly to its environment. A machine language program refers to an element of its environment by the address of the memory location or storage device containing the element. A JOVIAL program, on the other hand, refers to its environment by means of identifiers. A JOVIAL identifier, therefore, is a statement name, a switch name, a procedure name, an item name, a table name, or a file name.

An identifier is an arbitrary -- though usually mnemonic -- alphanumeric name, which serves to label a particular element in the environment of a JOVIAL program.

name ≠ letter:⌈letter;numeral⌉s⌊'⌋

Names may be constructed to suit the convenience of the programmer but, to enhance readability, should be as descriptive as possible. A name must start with a letter, followed by any number of letters or numerals, which may be punctuated for readability by the ' separator. (Since embedded blanks are not permitted, single primes may be used to connect multi-word names.) Some examples of JOVIAL names are given below.

STEPØ1

U2

BRANCH

FLIGHT'POSITION

A name may not have the same spelling as a delimiter, and may not end in a prime.

The scope of an identifier consists of the set of statements for which the identifier is defined. Within its scope, an identifier must have a unique spelling. However, statement names and switch names are distinguished by context from procedure names, item names, table names, and file names so that uniqueness between these two categories is not actually required. (Thus, for example, a statement name may duplicate an item name, though this is not a recommended practice.)

Except for statement names which are defined by context, all JOVIAL identifiers must be defined by a declaration of some sort so that the association of identifiers with environmental elements is decidable. These declarations may either be explicitly supplied by the programmer or implicitly supplied by a COMPOOL list of system declarations. Identifiers used but not defined in a procedure (or program) must be defined at some higher level, i.e., in the program (or COMPOOL).

Identifiers may be defined for just a single procedure, a single program, or an entire program system. Identifiers defined within a program or procedure are entirely local and do not conflict with identically spelled identifiers outside the program or procedure. Where such conflict might be thought to occur, the scope of the "outside" identifier excludes the scope of the "inside" identifier.

With these exceptions, the scope of an identifier naming a program element (statements, switches, and procedures) includes the entire procedure, program, or system; while the scope of an identifier naming a data element (items, tables, and files) includes just the statements listed after the defining declaration. (This means that data elements must be declared before they may be referenced.)

CONSTANTS

A constant denotes a particular data value that is unaffected by program execution. JOVIAL programs manipulate four types of data: numeric values, consisting of the class of rational numbers and rational number pairs; literal values, consisting of strings of JOVIAL signs; status values, consisting of independent sets of arbitrarily named states (such as Good, Fair, Poor); and Boolean values, consisting of the two values True and False. A JOVIAL constant, therefore, denotes a particular value as represented by a particular machine language symbol. Numbers, integers, and floating and fixed constants denote numeric values in the conventional, decimal sense; while octal constants have the obvious meaning of octal integers and dual constants denote pairs of numeric values. Literal constants denote JOVIAL sign strings, represented in one of two possible 6-bit-per-sign encoding schemes; status constants are mnemonic names denoting qualities or categories rather than numeric values; and Boolean constants denote either True or False.

constant ≠ integer-constant;floating-constant;fixed-constant;octal-

constant;dual-constant;literal-constant;status-constant;boolean-constant

A JOVIAL constant contains all the information needed by the compiler to perform the necessary constant to machine symbol conversion, and since machine symbols representing constant values are not duplicated, a single symbol may represent many different values.

NUMBERS. A number is a string of numerals denoting an unsigned, integral value: 98765432Ø1, for example.

number ≠ numerals

INTEGER CONSTANTS. Any integral value, positive or negative, may be denoted by an integer constant, composed of an optional + or - followed by a number. In the absence of a sign, the value is considered positive.

To avoid writing a lot of zeros, it is sometimes convenient to denote a very large integer as a coefficient multiplied by a positive, integral power of ten. Thus, both 25E9 and 25ØØØØØØØØØØ denote twenty-five billion. Because the base 1Ø is always the same, it is omitted in the integer constant and only the exponent need be written, separated from the coefficient by the abbreviation E for Exponent.

integer-constant ≠ ⌊+;-⌋:number:⌊E$_{xponent-base-1\emptyset}$:number⌋

Some examples of integer constants are given below.

27

-ØØ39

+331E9

FLOATING CONSTANTS. A floating constant (an optionally signed decimal number with a decimal point) denotes as its value a rational number with a floating-point machine language representation consisting of two binary symbols: the mantissa, a signed fraction representing the significant digits of the value; and the characteristic, a signed integer representing the exponent of an implicit power of two multiplier for the mantissa. Thus, the floating constant -5Ø. would be represented by a mantissa of -.781241 and a characteristic of +6, since

$-.781241 * 2^{6} = -5\emptyset.$

As with integer constants, it is often convenient to denote either a very large number or a very small number as a coefficient multiplied by an integral power of ten. Thus, both .1397E-8 and .ØØØØØØØØ1397 denote

$.1397 * 1\emptyset^{-8}$, and since $.1397 * 1\emptyset^{-8}$ equals $.15 * 2^{-3\emptyset}$

the floating constant .1397E-8 would be represented by a mantissa of +.15 and a characteristic of -3Ø.

Notice that a floating constant must have a decimal point.

floating-constant ≠ ⌊+;-⌋:⌈number:.:⌊number⌋⌉;⌈.:number⌉:⌊E$_{xponent-base-1\emptyset}$:

⌊+;-⌋:number⌋

Some examples of floating constants are given below.

27.

-Ø.Ø39

+3.31E-6

FIXED CONSTANTS. A fixed constant denotes as its value a rational number with a fixed-point machine language representation consisting of a single binary symbol: a signed or unsigned mixed number.

fixed-constant ≠ floating-constant:A:⌊+;-⌋:number$_{\text{of-fraction-bits}}$

The floating constant preceding the abbreviation A denotes the value, and the number following the abbreviation A indicates the precision of the value thus denoted by specifying the number of fractional bits (bits After the binary point) in the machine language symbol representing the value. This indication of precision determines how the machine language symbol is to be manipulated during arithmetic calculations, and also serves to distinguish fixed from floating constants.

If the value portion of a fixed constant denotes more precision than is indicated by the precision portion, the excess precision is truncated. Such truncation may be illustrated symbolically by considering r, the rational number actually denoted by the (positive) fixed constant vAp:

$r = v - v \text{ (modulo } 2^{-p})$

Thus, 27.98A1 and 27.5A1 both denote the same rational number, as do 2Ø7.A-3 and 2ØØ.A-3.

A negative precision indicates the number of least significant integral bits truncated, placing the binary point beyond the right end of the machine language symbol representing the value. A zero precision indicates an integral value (usually denoted by an integer constant).

Notice that a fixed constant must have a decimal point, the abbreviation A, and a precision indication.

Some further examples of fixed constants are given below.

-123.A4

5.5A5

+.678E9A-2Ø

OCTAL CONSTANTS. An octal constant, composed of a string of octal digits delimited by 0(and), denotes either an unsigned integral value, or a literal value (see the paragraph on literal constants).

octal-constant ≠ O_{ctal}:(:⌈ø;1;2;3;4;5;6;7⌉s:)

In denoting an integral value, an octal constant is useful in those cases where the programmer is more concerned with the bit pattern of the resulting machine language symbol representing the value than in the value itself. Since:

$ø_8 = øøø_2$

$1_8 = øø1_2$

$2_8 = ø1ø_2$

$3_8 = ø11_2$

$4_8 = 1øø_2$

$5_8 = 1ø1_2$

$6_8 = 11ø_2$

$7_8 = 111_2$

the bit pattern is readily discernible. For example: the bit pattern of the symbol representing the value denoted by O(2715346) is øø1ø111øø11ø1ø111øø11ø.

The integer value i denoted by the octal constant $O(o_n o_{n-1} \cdots o_2 o_1 o_ø)$ is given symbolically:

$$i = 8^n o_n + 8^{n-1} o_{n-1} + \ldots + 8^2 o_2 + 8 o_1 + o_ø$$

DUAL CONSTANTS. A dual constant, composed of a pair of integer, fixed, or octal constants separated by a comma and delimited by D(and), denotes as its value a rational number pair, each with a fixed-point machine language representation consisting of a single binary symbol: a signed (or unsigned) integer or mixed number. Both halves of a dual value must have the same precision.

dual-constant ≠ D_{ual}:(:⌈integer-constant:,:integer-constant⌉;

⌈fixed-constant:,:fixed-constant⌉;⌈octal-constant:,:octal-constant⌉:)

Dual constants are useful in calculations involving two-dimensional (x,y) coordinate systems. For example: D(+32.5øA5,-84.25A5) could denote

a location $32\frac{1}{2}$ miles east and $84\frac{1}{4}$ miles south of a sector center -- to a precision of 1/32nd of a mile. Some further examples of dual constants are given below.

D(ø,ø)

D(-165.9E-5A23,+89.1E-4A23)

D(0(177777),0(øøøøøø))

LITERAL CONSTANTS. A literal value is one identical to the symbol denoting it. A literal constant therefore, composed of a string of JOVIAL signs delimited by H(or T(and) and prefixed by the number of signs in the string, denotes as its value that selfsame string of signs, each sign represented in machine language by a 6-bit symbol.

literal-constant ≠ number$_{\text{of-signs}}$:H$_{\text{ollerith}}$;T$_{\text{ransmission-code}}$:(:signs:)

Literal constants are useful for denoting non-numeric values that can be conveniently represented by symbols constructed from the alphabet of JOVIAL signs, for example: words or phrases from a natural language such as English, or from a formal language such as JOVIAL itself. Since much of the communication between man and computer uses the JOVIAL alphabet, literal constants are especially useful in the processing involved in forming and interpreting such messages. Thus:

4øH(READY MORE INPUT AND RESUME COMPUTATION.)

could denote an output message for the computer operator.

Of the two binary coding schemes available for representing literal values, the more generally useful is Hollerith, the code by which literal values are input and output. Occasionally, however, the programmer is concerned with the exact form of the machine language representation of his literal values, and standard Transmission code, with its defined representation, is required.

OCTAL	TRANSMISSION-CODE	OCTAL	TRANSMISSION-CODE	OCTAL	TRANSMISSION-CODE
0(ØØ)	1T( )	0(25)	1T(P)	0(5Ø)	1T(*)
0(Ø6)	1T(A)	0(26)	1T(Q)	0(51)	1T(()
0(Ø7)	1T(B)	0(27)	1T(R)	0(56)	1T(,)
0(1Ø)	1T(C)	0(3Ø)	1T(S)	0(6Ø)	1T(Ø)
0(11)	1T(D)	0(31)	1T(T)	0(61)	1T(1)
0(12)	1T(E)	0(32)	1T(U)	0(62)	1T(2)
0(13)	1T(F)	0(33)	1T(V)	0(63)	1T(3)
0(14)	1T(G)	0(34)	1T(W)	0(64)	1T(4)
0(15)	1T(H)	0(35)	1T(X)	0(65)	1T(5)
0(16)	1T(I)	0(36)	1T(Y)	0(66)	1T(6)
0(17)	1T(J)	0(37)	1T(Z)	0(67)	1T(7)
0(2Ø)	1T(K)	0(4Ø)	1T())	0(7Ø)	1T(8)
0(21)	1T(L)	0(41)	1T(-)	0(71)	1T(9)
0(22)	1T(M)	0(42)	1T(+)	0(72)	1T(')
0(23)	1T(N)	0(44)	1T(=)	0(74)	1T(/)
0(24)	1T(O)	0(47)	1T($)	0(75)	1T(.)

Literal values (or rather, their machine language representations) may also be denoted by octal constants; in order, for example, to denote a code not associated with a JOVIAL sign. To illustrate: the machine language symbol representing 0(77) has no counterpart in the standard Transmission encoding of the JOVIAL alphabet, while both 0(371625) and 3T(ZIP) are represented in machine language by the same 18-bit symbol.

It is important to note that any JOVIAL sign may appear in a literal constant's sign string, including parentheses and blanks, so that the number of signs within the delimiting parentheses must be exactly equal to the number preceding the abbreviation H or T.

STATUS CONSTANTS. A status constant, composed of a letter or name delimited by V(and), denotes one of a set of arbitrarily labeled values.

status-constant ≠ V$_{alue}$:(:letter;name:)

Each value in such a set is denoted by its own unique and usually mnemonic status-constant, and is represented in machine language by a single binary integer. For example, playing card values might be denoted by the following set of status constants:

V(JOKER) V(ACE) V(DEUCE) V(TREY) V(FOUR) V(FIVE) V(SIX) V(SEVEN) V(EIGHT) V(NINE) V(TEN) V(JACK) V(QUEEN) V(KING)

and be represented by binary symbols equivalent to: Ø for Joker; 1 for Ace; 2 for Deuce; 3 for Trey; and so on, to 13 for King.

Since a particular status constant may denote several different values from several different sets, the correspondence between status constant and integer depends on context.

BOOLEAN CONSTANTS. A Boolean constant denotes one of the logical values, True or False, of Boolean algebra -- as represented in machine language, either non-zero for True or zero for False.

boolean-constant ≠ $1_{\text{denoting-true}}$;$\emptyset_{\text{denoting-false}}$

Boolean constants may also be used to denote other pairs of dichotomous values, for example: Yes/No; On/Off; Minus/Plus; Set/Reset; In/Out; Friend/Foe; and so on.

EXERCISE (Constants)

(a) Construct a fixed constant denoting the value of $\pi = 3.1415926536\ldots$ to a precision of at least six decimal places.

(b) Construct a fixed constant denoting the value of $e = 2.7182818285\ldots$ to a precision of at least three decimal places.

(c) Express the rational number denoted by the fixed constant 81.5653Ø4A15 in floating constant form as a fraction with a power of ten multiplier.

(d) A sector has a radar site 89 miles NNW of the sector center. Express this location, to a precision of ± .Ø5, as a dual constant.

(e) Construct an octal constant represented by the same machine language symbol representing 6T(JOVIAL) and (f) construct an integer constant represented by this symbol.

(g) The value 'Excellent' may be denoted by either 9H(EXCELLENT) or V(EXCELLENT) and, as denoted by V(EXCELLENT), is typically represented by a three-bit machine language symbol. What advantage, if any, is there in denoting such a value by a status rather than by a literal constant, and (h) what are the disadvantages?

(i) A set of values consists of the two colors red and black. Construct a pair of JOVIAL constants denoting these values.

(j) A set of values consists of the four playing card suits. Construct a set of JOVIAL constants denoting these values.

COMMENTS

A comment, composed of an arbitrary string of JOVIAL signs delimited by '' and '', allows the inclusion of clarifying text among the symbols of a program. Comments are treated as strings of blanks by the compiler and are thus ignored, having no operational effect whatever on the program

blanks ≡ ''signs$_{\text{except-the-symbols-''-and-\$}}$''

Comments should be used generously nevertheless, since careful annotation is necessary even when programs are coded in as readable a language as JOVIAL -- for readability does not necessarily imply understandability, and as much care should be taken to make a program understandable as must be taken to make it workable.

Comments aid both the original programmer, who often tends to forget the precise function of parts of his program after they have grown "cold," and his successors, who must complete or maintain or modify the program. The programmer's intimate and detailed knowledge of his program and of the problem it solves is more valuable than the program itself, and much of this knowledge can be recorded as program comments. The importance of adequate commentary cannot therefore be over-emphasized.

A comment may be inserted between any pair of symbols in a JOVIAL program.* This makes it possible to write JOVIAL sentences in a rather spasmodic prose style that resembles English. For example:

```
IF''the current value of item''NTRK'', the number of tracks in sector''
($S$)''is''GR''eater than''31 $
     ''Then''GOTO''process''EXCESS''tracks for sector''($S$)
     ''as described on pages 17...21 of this listing''$
''Otherwise, set the''SITU''ation status item for sector''($S$) = V(FEW) $
```

illustrates the mechanics of inserting comments. Stripped of commentary, the routine reads:

---

* Except within a DEFINE declaration.

```
IF NTRK($S$) GR 31 $  GOTO EXCESS($S$) $  SITU($S$) = V(FEW) $
```

which is actually not quite so cryptic as at first glance it might seem. Comments at the above level of detail are rarely necessary unless the program must be read by non-programmers, since it is usually not individual statements that need explanation but groups of statements. (With a little practice, the function of any simple JOVIAL statement is almost immediately obvious.)

The text portion of a comment must include neither '', the double prime bracket, (since this, of course, ends the comment) nor $, the terminating separator (which is used exclusively for terminating JOVIAL sentences). Further, the omission of either delimiting double prime is a major error, for subsequent commentary is interpreted by the compiler as program, and vice-versa. The results of compiling English language comments are not usually very rewarding.

CLAUSES

Strings of JOVIAL symbols (delimiters, identifiers, and constants), separated by blanks that may be omitted where this does not join a numeral/letter pair, form clauses, which are: item descriptions; variables; and formulas. An item description describes a value; a variable designates a value; and a formula specifies a value.

ITEM DESCRIPTIONS

In JOVIAL, the basic units of data are called items. All the necessary characteristics of an item's value, such as its type, and the format and coding of the machine symbol representing it, need be supplied only once, in an item description. Item descriptions for the various types of value will be discussed separately, in the paragraphs on numeric, literal, status, and Boolean clauses.

VARIABLES

A JOVIAL variable designates a value which may vary during the course of program execution. (One of the main functions of any program is to compute fresh values for the variables in its environment.) Varying the value of a variable is accomplished by replacing the machine symbol representing the old value with another representing the new value. A symbol may be thus replaced by the assignment of a specified symbol in its place, by exchange with another symbol, or by the input of a record from a file.

There are four major types of variable in JOVIAL, corresponding to the four basic types of data: numeric; literal; status; and Boolean. (A fifth type of variable, the entry variable, designates a composite value and is thus in a category by itself. Entry expressions will be discussed later, along with the ENTRY functional modifier.) As basic units of data, items are the principal variables of JOVIAL. A JOVIAL variable is therefore usually composed of an item name -- possibly suffixed by an index delimited by the subscription brackets ($ and $).

variable ≠ name$_{\text{of-item}}$ ⌊($ index $)⌋

An index* subscripting an item name distinguishes a particular value from a set of values bearing the same item name. For example,

ALPHA($Ø,13,9,2$)

designates the value of the item ALPHA for row Ø, column 13, plane 9, and space 2 of a 4-dimensional array of ALPHAs. Similarly,

BETA(I)

designates the value of the item BETA for entry I of some table; while

GAMMA

designates the value of a simple, "one-of-a-kind," unsubscripted item named GAMMA.

FORMULAS

A JOVIAL formula specifies a single data value and is, in effect, a rule for obtaining that value -- perhaps by means of a lengthy computation. Formulas are classified according to the type of data value they specify: numeric; literal; status; and Boolean. They are composed of operands, which are constants, variables, and functions, and of arithmetic, relational, or logical operators. A numeric or Boolean formula may specify a value in terms of a series of operations upon a set of operands, while a literal or status formula must specify a value in terms of a single operand.

---

* Indexes will be more fully discussed in a later paragraph.

formula ≠ numeric-formula;literal-formula;status-formula;Boolean-formula

FUNCTIONS. A function specifies a single data value and is composed of a procedure name followed by a list of calling parameters, which are either formulas or names, separated by commas and bracketed by the (and) parentheses. A parameterless function is possible, but the parentheses may not be omitted.

function ≠ name$_{\text{of-procedure}}$ (⌊⌈formula;name⌉s'⌋)

Functions are numeric, literal, status, or Boolean -- according to the type of data value they specify. The value specified by a function is computed by the procedure, which is automatically invoked whenever the function's value is needed. In its execution, the procedure utilizes the values specified by the calling parameter formulas and the environment elements denoted by the calling parameter names. (Both calling parameters and the computation of function values by procedures will be discussed later, in the section on procedures.)

Functions serve much the same purpose in JOVIAL that they do in ordinary mathematics. That is: a function allows the specification of a parameter dependent value to be removed from the description of how the value is derived. Thus, in mathematics, once the cosine function is defined, the term cos θ may be used wherever it is required. Similarly, in JOVIAL, if an appropriate procedure named COS is defined, the function

COS (THETA)

would specify the cosine of the angle given by the value of the item THETA. Some further examples of functions are given below.

ARCSIN (GAMMA+2.72,1.ØE-4)

RANDOM ()

SYMMETRIC (MATRIX'A)

BASIC NUMERIC CLAUSES

A numeric clause describes or designates or specifies a numeric value, one that may be denoted by an integer, floating, fixed, dual, or

octal constant. A floating, fixed, or dual item description describes a numeric value (or rather the machine symbol representing the value) while a numeric variable designates and a numeric formula specifies a numeric value.

The discussion of numeric clauses will be based on the following definitions:

. Precision -- the number of digits with which a numeric value is expressed. A convenient measure of precision is the minimum value expressible in the same number of digits.

. Significance -- the minimum number of digits with which a numeric value could be expressed without loss of accuracy.

. Accuracy -- the number of 'correct' digits with which a numeric value is known. A measure of accuracy is maximum error.

For example: the number ØØØ19.3416Ø has 6 or 7 significant digits (trailing zeros may not be significant, leading zeros never are) and is precise to .ØØØØ1, but may only be accurate to ± .ØØ5. Obviously, accuracy cannot exceed significance, and significance cannot exceed precision.

NUMERIC ITEM DESCRIPTIONS

A numeric item description describes the machine symbol used to represent the value of a floating, fixed, or dual item. Two optional elements are common to all three descriptions:

1. The Rounded descriptor, which declares that any value assigned to the item be rounded to the required precision rather than truncated, as would otherwise be the case. (Rounding is accomplished by first adding 2^{p-1} to the magnitude of the value being assigned to the item, where 2^p is the value that can be represented by the magnitude of the item's least significant bit. The symbol representing the resulting sum is then truncated to the precision of the item, replacing the symbol representing the item's old value. Thus, the value 4.75 assigned to a rounded integer item becomes 5, and assigned to a truncated integer item, the same value becomes 4.) Rounding is generally useful where it is desired to squeeze the maximum accuracy out of computations with a limited precision.

2. A pair of constants, separated by the ... separator, which declare, in order, the estimated minimum and maximum absolute values of the item. (This optional magnitude range may be used by a JOVIAL compiler to optimize the machine language program's manipulation of the item's value.)

FLOATING-POINT ITEM DESCRIPTIONS. A floating-point item description, composed of the type descriptor Floating, followed by the optional Rounded descriptor, followed by an optional pair of floating constants separated by the ... separator, is used to declare an item whose arithmetic value is represented by a floating-point machine symbol.

description$_{\text{of-floating-point-item}}$ ≠ Floating ⌊Rounded⌋ ⌊floating-constant ... floating-constant⌋

- The type descriptor Floating declares the item a floating-point type item.

o The Rounded descriptor declares the item's value to be rounded.

o The optional pair of floating constants declare an estimated absolute value range.

Some examples of floating-point item descriptions are given below.

Floating

Floating Rounded

Floating ∅.1E-2∅...∅.5E+2∅

Floating Rounded ∅. 9E-9

FIXED-POINT ITEM DESCRIPTIONS. A fixed-point item description, composed of the type descriptor fixed (which may be abbreviated by A), followed by number of bits, either the Signed or Unsigned descriptor, an optional number of fraction bits, the optional Rounded descriptor, and an optional pair of integer or fixed constants separated by the ... separator, is used to declare an item whose numeric value is represented by a fixed-point machine symbol.

description$_{\text{of-fixed-point-item}}$ ≠ fixed number$_{\text{of-bits}}$ Signed;Unsigned ⌊⌊+;-⌋:number$_{\text{of-fraction-bits}}$⌋ ⌊Rounded⌋ ⌊integer-constant;fixed-constant ... integer-constant;fixed-constant⌋

• The type descriptor fixed (or the abbreviation A) declares the item a fixed-point type item.

• The number of bits declares the total number of bits in the item's machine symbol, including any sign bit.

• The Signed descriptor declares a signed value, with one of the bits of the machine symbol serving as a sign-bit. The Unsigned descriptor declares an unsigned value that is always positive.

o The optional, signed number of fraction bits declares the number of fractional bits in the item's machine symbol. A zero indicates an integral value and may be omitted for exact integers. A number equal to the number of magnitude bits (total number of bits if unsigned, total minus 1 if signed) indicates a purely fractional value. A number greater than the number of magnitude bits indicates that the first few fraction bits of the value are not significant and thus need not be represented. A negative number indicates that the last few integer bits of the value are not significant and thus need not be represented.

o The optional Rounded descriptor declares the item's value to be Rounded.

o The optional pair of integer or fixed constants declare an estimated absolute value range.

The following examples illustrate the effect of a fixed-point item description on the format of the machine symbol being described. (Where s is a sign-bit, i is an integer-bit, f is a fraction-bit, and - is an implied bit, not included in the machine symbol.)

Description	Format
fixed 7 Signed	s i i i i i i.
fixed 7 Unsigned 2	i i i i i.f f
fixed 7 Unsigned 9	.- - f f f f f f f
fixed 7 Signed -2	s i i i i i i - -.

In designing a fixed-point item description, certain information must be available about the value being described, mainly: the precision with which it need be represented; and its range of variation. Precision determines the number of fractional bits required while range determines the number of integral bits required, as well as whether the value need be signed or not.

For example: suppose the variable ALPHA ranges in value between ø and 1øø, and requires .ø1 precision. Since:

$$2^6 = 64 < 1øø < 128 = 2^7$$

it can be seen that 7 integral bits are needed to represent the maximum value of ALPHA. Similarly, since:

$$2^{-7} = .øø781 < .ø1 < .ø156 = 2^{-6}$$

it can be seen that 7 fractional bits are needed to provide the desired precision. And since:

ø < ALPHA

the machine-symbol representing ALPHA need not include a sign-bit. These considerations lead to the conclusion that a machine-symbol with the format

i i i i i i i.f f f f f f f

will suffice to represent all values of ALPHA. A fixed-point item description for this format is

fixed 14 Unsigned 7 Rounded ø...1øø

which may be expanded with comments to

fixed''point''14''bits''Unsigned''7''fraction bits''Rounded''range''ø...1øø

The following examples further illustrate the way in which fixed-point item descriptions are designed.

Consider the item YTD'NET, year-to-date net earnings, from a typical personnel file. Assume this item is measured in dollars and must be precise to the nearest penny. As before, a precision of .ø1 requires 7 fractional bits. If the annual salary of the firm's president is $1øø,øøø, then 17 integral bits will suffice to represent any earnings, since:

$$2^{16} = 65536 < 1øøøøø < 131ø72 = 2^{17}$$

As net earnings mean <u>after</u> deductions, negative values are conceivable so the item must be signed. In short:

fixed 25 Signed 7 ø.A7 ... 1.E5A7

Consider the item ALTITUDE used in processing aircraft flight data. ALTITUDE is measured in feet but need be precise only to the nearest 1ØØ foot interval. Since:

$$2^6 = 64 < 1ØØ < 128 = 2^7$$

we find that, far from needing any fractional bits, the precision requirement allows us to dispense with the 6 least significant integral bits. This precision is declared as -6 fraction-bits. A comfortable 1ØØ,ØØØ foot ceiling means that, as before, 17 integral bits are needed. Chopping off the 6 least significant bits to agree with the precision requirement yields a total of 11 magnitude bits, and since ALTITUDE is implicitly positive, it needs no sign. A fixed-point machine symbol representing ALTITUDE would therefore have the format

i i i i i i i i i i i - - - - - - .

with the binary point 6 positions past the right end of the symbol. In short:

fixed 11 Unsigned -6

If, however, the value of ALTITUDE must be precise to the nearest foot, then

fixed 17 Unsigned

describing a 17-bit, unsigned integer is needed instead.

Consider the item DELTA from an engineering problem, where:

$$-.ØØØØ5 < \text{DELTA} < +.ØØØØ5$$

Since:

$$-2^{-14} < -.ØØØØ5 < \text{DELTA} < +.ØØØØ5 < +\ 2^{-14}$$

the absolute magnitude of DELTA is less than 2^{-14}, indicating that the required number of integral bits is -14, which means that the 14 most significant fractional bits are unnecessary and may therefore be omitted. If DELTA must be precise to $\pm\ 1Ø^{-9}$, then 3Ø fractional bits are required, since:

$$2^{-30} < 1Ø^{-9} < 2^{-29}$$

To represent both positive and negative values of DELTA, a sign-bit is needed, so that the total number of bits required is the sum of (the number of integral bits) -14, (the number of fractional bits) 3Ø, and (the sign-bit) 1. This yields a total of 17, so that DELTA must be represented by a machine-symbol with the format

s.- - - - - - - - - - - - - - - f f f f f f f f f f f f f f f

which may be described as follows:

fixed 17 Signed 3ø Rounded

FIXED VS. FLOATING REPRESENTATION. A fundamental problem in numerical calculations is that of maintaining significance in the values involved.

To maintain significance using fixed-point machine symbols, it must always be possible to predict the maximum size of the variable values being represented, and often the minimum size as well. For some problems, however, it is extremely difficult or even impossible to make such predictions. Or where prediction is possible, the variation in size may be so great that a very large machine symbol would be required to retain significance over the entire range. Consequently, fixed-point representation is practical only for those variables with a limited and predictable range in size.

Floating-point machine symbols, on the other hand, automatically maintain the 8 or 1ø most significant decimal digits of values typically ranging in size between 1.E-3øø and 1.E+3øø. (Particular limits are, of course, machine dependent.) Such symbols are also much easier to describe than fixed-point machine symbols since only one format is possible. They are, however, less efficiently manipulated by the computer than fixed-point symbols and are not really suited for the representation of exact values, such as integers, which must be specifically precise. In addition, floating-point computations can produce violent and unexpected arithmetic errors unless thoroughly analyzed. For example:

$$(.56789ø1234*1ø^{1øø} - .56789ø1123*1ø^{1øø}) * .98765ø1234*1ø^{7} = .1ø96291637*1ø^{1øø}$$

The result of this computation with decimal floating-point symbols is clearly not accurate to 1ø digits as shown, because the subtraction lost 7 digits.

DUAL ITEM DESCRIPTIONS. A dual fixed-point item description, composed of the type descriptor Dual, followed by number of bits per component, either the Signed or Unsigned descriptor, an optional number of fraction bits per component, the optional Rounded descriptor, and an optional pair of dual constants separated by the ... separator, is used to declare an item whose dual numeric value is represented by a dual, fixed-point machine symbol.

description$_{\text{of-dual-item}}$ ≠ Dual number$_{\text{of-bits}}$ Signed;Unsigned ⌊⌊+;-⌋: number$_{\text{of-fraction-bits}}$⌋ ⌊Rounded⌋ ⌊dual-constant ... dual-constant⌋

• The type descriptor Dual declares the item a dual, fixed-point type item. The remainder of the description applies to each of the two component halves of the item and has the same meaning as the corresponding elements in the fixed-point item description.

• The number of bits declares the total number of bits in the machine symbol representing a single component of the dual value, including any sign-bit.

• The Signed descriptor declares a signed component, and the Unsigned descriptor declares an unsigned component.

o The optional, signed number of fraction bits declares the number of fractional bits in the machine symbol representing a component value. Both components thus have the same precision.

o The optional Rounded descriptor declares both components' values to be rounded.

o The optional pair of dual constants declare an estimated absolute value range.

Dual item descriptions are designed in much the same way as are fixed item descriptions. For example: consider the item AIRCRAFT'POSITION, expressed in x,y coordinates measured in nautical miles from a sector center. AIRCRAFT'POSITION must be precise to .ø5 miles, and aircraft further than 1øøø miles from the sector center are not recorded. AIRCRAFT'POSITION is a dual value requiring 1ø integral bits in each component to represent values as great as 1øøø, since:

$2^9 = 512 < 1øøø < 1ø24 = 2^{1ø}$

Aircraft can be East or West, North or South of the sector center, so the component values must be signed, and since:

$2^{-5} = .ø3125 < .ø5 < .ø625 = 2^{-4}$

each component requires 5 fractional bits, making a total of 16 bits per component, for a grand total of 32 bits. The item description

Dual 16 Signed 5 Rounded

describes a machine symbol with a format

s i i i i i i i i i.f f f f f,s i i i i i i i i i.f f f f f

meeting the specifications.

EXERCISE (Numeric Item Descriptions)

(a) Design an item description for a variable with a range -1.E2 thru +1.E6, which must be precise to ± .ØØØ5.

(b) Design an item description for a variable with a range 16 thru 8ØØØØ, which must be precise to the nearest integer.

(c) Design an item description for a variable with a range -1.E1ØØ thru +1.E1ØØ, where exact precision is unimportant.

(d) Discuss the variable with the following item description in terms of range and precision: fixed 17 Signed 2Ø

(e) Design an item description and give the units of measure, for a variable whose value must locate any point on the surface of the earth to a precision of ± .Ø5 nautical miles. (Hint: use latitude and longitude. 1° of arc at the equator = 6Ø nautical miles.) Also, construct a constant denoting the location, in the vicinity of New Orleans, at $3Ø^{\circ}N, 9Ø^{\circ}W$.

NUMERIC VARIABLES

A numeric variable designates either a rational number as represented by a fixed or floating-point machine symbol, or a rational number pair, as represented by a dual fixed-point machine symbol.

Floating, fixed, and dual items (both simple and subscripted by an index) are thus numeric variables. For example:

ALPHA

YTD'NET($EMPLOYEE'NR$)

ALTITUDE(F)

DELTA($I+1,J,K-1$)

AIRCRAFT'POSITION(F)

are all numeric variables, designating values described in the preceding sections.

A single-letter subscript is also a numeric variable, designating a signed, integral value. Subscript variables may be considered as elements of the language, because they are implicitly described. Subscript description varies from machine to machine, but the basic format may be thought of as

s i i i ... i i i.

that is, a sign-bit and an undefined number of integer bits. (A subscript will always suffice to index the greatest array that may be stored in the computer's memory.) Subscripts are activated (i.e., defined and assigned an initial value) by FOR statements, and though as many as 26 subscripts (A thru Z) may be active simultaneously, a subscript's range of activity is strictly limited. (See the section on FOR statements.)

Other numeric variables, involving functional modifiers, will be discussed later.

NUMERIC FORMULAS

A numeric formula specifies a single, numeric value, obtained by executing any indicated arithmetic operations on the values of the numeric operands composing the formula. These operand values may be: denoted by integer, floating, fixed, octal, or dual constants; designated by arithmetic variables; or computed as function values by procedures.

numeric-formula ≠ integer;floating-constant;fixed-constant;octal-constant;

variable$_{\text{of-numeric-type}}$;function$_{\text{of-numeric-type}}$;⌈(numeric-formula)⌉;

⌈(/ numeric-formula /)⌉;⌈⌊+;-⌋ numeric-formulas⌈+;-;*;/;**⌉⌉

A JOVIAL numeric formula containing arithmetic operators specifies a value in much the same way as would a similar formula in the language of ordinary algebra. Thus, the arithmetic operators +, -, *, and / have the expected meanings of addition, subtraction or negation, multiplication, and division; while the arithmetic operator ** has the, perhaps, not so obvious meaning of exponentiation, raising to a power.

As in algebra, division by zero is undefined. Fractional or mixed exponents are possible, but since JOVIAL deals only with rational numbers, any exponentiation which would specify a complex root in algebra, such as (-2**.5), is also undefined.

The parentheses (and) perform their usual grouping function, and the absolute value brackets (/ and /) specify the magnitude of the value of the numeric formula they enclose. With these brackets, formulas of any complexity may be constructed.

Consider the familiar algebraic formula:

$Ax^2 + Bx + C$

An equivalent JOVIAL formula is:

```
AA*XX**2 + BB*XX + CC
```

Note that, unlike algebra, multiplication must be explicitly indicated in JOVIAL and also, though algebra customarily uses a single letter to designate a value, JOVIAL customarily uses multi-character identifiers since JOVIAL is restricted in alphabet while algebra is not.

The main difference, however, between a JOVIAL numeric formula and an algebraic formula arises from the fact that JOVIAL is a linear language: everything must be strung out as on a single line. As a result, JOVIAL requires special symbols for subscripting and superscripting, and a JOVIAL arithmetic formula will often contain a profusion of parentheses where none are required in algebra. For example, the algebraic formula:

$$\frac{A_{\emptyset}^2}{A_1 - \dfrac{A_2}{x + \dfrac{1}{x}}}$$

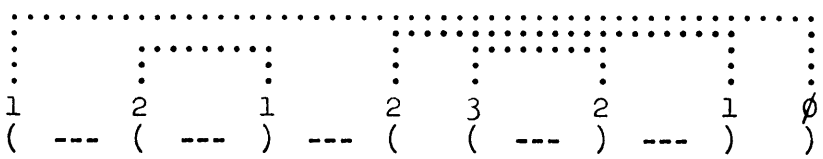
could appear in JOVIAL as:

```
(AA($Ø$)**2)/(AA($1$)-(AA($2$)/(XX+(1/XX))))
```

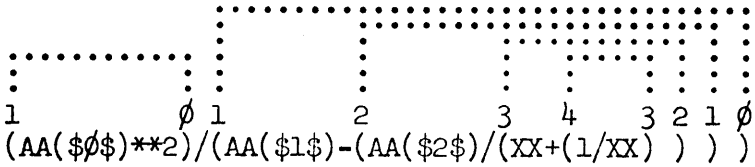
While algebra customarily uses many kinds of brackets to group sub-formulas, JOVIAL is limited to (and), which sometimes makes it difficult to mate left and right parentheses properly. A method for finding parenthesis pairs is to number each parenthesis from left to right, increasing the count by one for each left parenthesis and decreasing it by one for each right parenthesis. The number of the last parenthesis in the formula should be zero. The mate to a left parenthesis is the first right parenthesis with a smaller number. Thus,

```
( --- ( --- ) --- (   ( --- ) --- )   )
```

would be numbered and paired as follows:



Performing this analysis on the parentheses in the previous formula



shows that the sub-formulas have been acceptably grouped. Actually, not all the parentheses used in this example are necessary, and the formula could appear as

```
AA($ø$)**2/(AA($1$)-AA($2$)/(XX+1/XX))
```

Three pair of parentheses are redundant because of the implicit ordering of operations in a JOVIAL numeric formula.

SEQUENCE OF NUMERIC OPERATIONS. The sequence of operations described in a numeric formula is determined primarily be the way the formula is bracketed, and secondarily by an operator precedence scheme. Operations within sub-formulas, enclosed in parentheses or the absolute value brackets (/ and /), are performed first. Thus, 2*((6+3)/3) specifies 6 as follows: (6+3) specifies 9; (9/3) specifies 3; and 2*3 specifies 6. Any particular operator may thus be given precedence by bracketing it and its operands. Where bracketing does not unambiguously indicate precedence, (i.e., in sub-formulas with more than one operator) the conventional rules apply:

A. Negations are performed first. Thus, -2+3 specifies +1 and not -5. (The unary* operator + is allowed, but is redundant, ignored, and usually omitted except for emphasis, as +1 in the previous sentence.) In a bracketless formula, negations either come first or follow another operator, for example: ALPHA/-9.7E6.

B. Exponentiations are performed next. Thus, 2*2**2 specifies 8, not 16; and 2+2**2 specifies 6, not 16.

C. Multiplications and divisions are performed before additions and subtractions. Thus 2+2/2 specifies 3, not 2; and 2-2*2 specifies -2, not ø.

---

* A unary operator works on a single operand, unlike a binary operator, which works on two.

D. Additions and subtractions are performed last.

E. Within the above categories, operations are performed from left to right in order of listing. Thus: 2**3**2 specifies 64, not 512; 2./2./*2. specifies 2., not .5; and 2./2./2. specifies .5, not 2.

The sequence of operations given by these rules of precedence can, of course, be altered by the use of additional parentheses. And it is often desirable to emphasize a correct sequence of operations with redundant parentheses, to avoid possible misunderstanding.

ALGEBRA OF NUMERIC OPERATIONS. By algebraic manipulation, it is often possible to rearrange a numeric formula to yield a more efficient scheme of operations. Constant terms, for example, should be combined. This easy-to-correct inefficiency is illustrated in the formula

```
3.*ALPHA/1.E6
```

which could be rewritten as

```
ALPHA/333333.3333
```

to contain only one operation and require storage for only one constant machine symbol.

The factoring-out of common terms can also reduce the number of operations to be performed. To illustrate: the algebraic formula used previously,

$Ax^2 + Bx + C$

was written in JOVIAL as

```
AA*XX**2 + BB*XX + CC
```

The formula indicates five arithmetic operations, which by simple algebra

$Ax^2 + Bx + C = (Ax + B)x + C$

could be reduced to four, thus:

```
(AA*XX + BB)*XX + CC
```

This particular type of factorization is called 'nesting', for a reason which requires a slightly more ambitious polynomial to illustrate.

Consider then:

$$A_4x^4+A_3x^3+A_2x^2+A_1x+A_\emptyset = (((A_4x+A_3)x+A_2)x+A_1)x+A_\emptyset$$

In the right-hand formula, successive factors are embedded in 'nests' of parentheses, thus 'nesting'. When transliterated to JOVIAL, the nested formula

```
(((AA($4$)*XX+AA($3$))*XX+AA($2$))*XX+AA($1$))*XX+AA($Ø$)
```

specifies only eight operations, as against the dozen or so that the unnested version would have required.

The process of transliterating a formula from algebra into JOVIAL is shown in the following examples:

Algebra	JOVIAL
$\frac{1}{3}A + \frac{4}{3}B + \frac{1}{3}C$	AA/3.+4.*BB/3.+CC/3. or better: AA/3.+1.333333333*BB+CC/3. or better yet: (AA+4.*BB+CC)/3.
$\frac{5x+4}{2x+8} + \frac{1}{2}$	(5.*XX+4.)/(2.*XX+8.) + .5 or better: (3.*XX+4.)/(XX+4.)
$(\frac{1}{2}(A+B+C)(B+C)(A+C)(A+B))^{\frac{1}{2}}$	(.5*(AA+BB+CC)*(BB+CC)*(AA+CC) *(AA+BB))**.5
$\frac{\lvert x - A\rvert^{\frac{1}{2}} + (x^2 + A)}{A^{3/2}}$	((/XX-AA/)**.5+(XX**2+AA))/AA**1.5
$\frac{x^3(x+2)}{3x^{\frac{1}{4}}} - (3x)^{\frac{1}{2}}$	(XX**3*(XX+2))/(3.*XX**.25)-(3.*XX)**.5
$(\alpha_i{}^i)^2$	(ALPHA(I)**I)**2
$\alpha_{i-1}^{i^2}$	ALPHA($I-1$)**(I**2)

MODES OF NUMERIC OPERATION. In JOVIAL, numeric values have three modes of representation: floating-point; fixed-point; and dual fixed-point. Any arithmetic operation, upon operands of like mode, may be performed in any one of these modes. However, a numeric formula may combine operands having different modes of representation, so that automatic conversion between modes is implied.

Mode conversion is best described in terms of three fictitious functions: Float; Fix; and Twin. Float converts a value in fixed-point representation to floating representation. Thus:

Float(2) is 2.

Fix, on the other hand, converts a value in floating-point representation to fixed representation of an undefined precision. Thus:

Fix(2.) is 2.A?

Twin, finally, converts a value in fixed-point representation to dual representation by the simple expedient of duplicating the value, for example:

Twin(2) is D(2,2)

Twin(Fix(2.)) is D(2.A?,2.A?)

Mode selection for an operation is based primarily on the mode of the operands involved, so that an operation on two operand values of like mode will be performed in that mode and will yield a resulting value in that mode. In the floating-point mode, for example: 4.5 + 2. specifies 6.5, and in the fixed-point mode: 4.5A3 * 2.A3 specifies 9.A3. In the dual fixed-point mode, operations are done in parallel, with the left component of one operand combined with the left component of the other to yield the left component of the result, and similarly for right components. Thus: D(4.5A3,9.A3) - D(2.A3,-2.A3) specifies D(2.5A3,11.A3); and D(4.5A3,9.A3)**D(2,2) specifies D(2Ø.25A3,81.A3).

When a dual operand is combined with a floating or a fixed operand, the mode of operation and the result are dual, with the implicit conversion Twin -- and if necessary, Fix -- employed on the mono-valued operand. Thus: D(4.5A3,9.A3)/2. specifies D(2.25A?,4.5A?); and 4.5A3**D(2.A3,.5A3) specifies D(2Ø.25A3,2.12A3).

When a fixed operand is combined with a floating operand, the mode of operation depends on the intended use of the result, as determined by the context of the formula. A mode of operation is selected which minimizes the number of Fix and Float conversions necessary in evaluating the formula, so that 4.5A3*2. could specify either 9. or 9.A?.

Note: For the sake of program efficiency, arithmetic formulas should be written to minimize the number of implicit Fix or Float conversions needed. (The minimization of Twin conversions in a formula is usually less important, because of the trivial nature of the conversion.) However, when any value, floating, fixed or dual, is to be raised to an integral power, an integer valued exponent is preferable, since it results in a more efficient program. In other words, though 3.**2. and 3.**2 both specify 9., the latter formula, with the integer exponent, is better.

PRECISION OF NUMERIC OPERATIONS. The computations performed in carrying out the intent of a JOVIAL numeric formula may possibly vary, in detail, for different computers. In particular, the representation and manipulation of negative values and the method of carrying signs will very likely differ. Another and more important point of difference is the precision with which computations are performed.

The precision* of the result of a floating-point computation cannot be defined, since it depends both on the magnitude of the value (which, of course, varies) and on the length of the mantissa portion of the floating-point machine symbol representing the value (which, of course, is computer dependent.) Where C is the value of the characteristic and n is the number of bits in the mantissa, the precision of a floating-point number is 2^{C-n} or, as JOVIAL would say

2. ** (CHARACTERISTIC - NUMBER'OF'MANTISSA'BITS)

The precision of the result of a fixed-point computation also cannot be exactly defined, but some useful limits can be established. (These limits also apply to dual fixed-point results.) The problem of fixed-point precision arises because, although the significance of the result of a fixed-point arithmetic operation can exceed the significance of the most significant operand (e.g., multiplication -- where an m-bit multiplier and an n-bit multiplicand can produce an mn-bit product) the accuracy of the result cannot exceed the significance of the least significant operand, while the number of bit positions available for the result is usually limited to the least multiple of the computer's

---

* In floating-point computations, it is ordinarily not precision which is of interest, but accuracy. Computational accuracy is established by a 4-step error analysis, determining: first, the probable initial error in the data; second, the local error arising from each computation; third, the propagated error as particular computations are repeated; and finally, the cumulative error for the total computation. Error analysis is not simplified by floating-point representation.

word size that may contain the most significant operand. Where the maximum significance of the result exceeds this limit, a precision must be selected so the result may be truncated with a minimum sacrifice of accuracy. Such truncation is performed in the following manner: first, the least significant fraction bits of the result are truncated; and second, if necessary, the most significant integer bits.

To give a decimal example of this truncation rule, suppose two variables are multiplied whose maximum magnitudes are 999.9999 and 9.9999 and whose product is limited to ten significant digits. Since the actual product could have as many as four significant integral digits and eight significant fractional digits, the two least significant fractional digits would be truncated, and the maximum product would be carried as 9999.899ØØØ, with six fractional digits rather than eight. As another example, if two variables have maximum magnitudes of 999999.9 and 99999., then their product would have as many as eleven significant integer digits and one significant fractional digit, or twelve significant digits in all. If their product is again limited to ten significant digits, then both the fractional digit and the most significant integral digit would be truncated. Such truncation introduces the possibility of spurious results, which must be carefully guarded against.

Note: Because of the unavoidable differences introduced by the truncation of intermediate results, not all algebraic identities are necessarily JOVIAL arithmetic identities. In algebra, for example: A+(B+C) ≡ A+B+C and (A+B)/C ≡ A/C+B/C. In JOVIAL, however: AA+(BB+CC) does not necessarily specify the same value as AA+BB+CC; nor does (AA+BB)/CC necessarily specify the same value as AA/CC+BB/CC. (Such truncation errors, of course, apply to all computations, and not just those involving fixed-point operands.)

EXERCISE (Numeric Formulas)

(a) Assuming I and N are active subscripts, designating the algebraic values i and n, and that AA, BB, CC, and XX are floating-point items designating the algebraic values A, B, C, and x, transliterate the following algebraic formulas into JOVIAL numeric formulas.

1. $\frac{1}{4}(x^n_{i+1} + x^n_i + x^n_{i-1})$

2. $A_i x_\emptyset + B_i x_1 + C_i x_2$

3. $(x^n)^{2i^2}$

4. $\dfrac{Ax_n^5 + (AC - B)x_n^3 + Cx_n^2 - |BCx_n - C^2|}{x_n^2 + C}$

5. $x^3 - \frac{3}{5}(A^2|x|^{\frac{1}{2}})$

6. $(A_n^2 + B_n^2 + C_n^2)^i$

7. $x^{(i^2 + n^2)}$

8. $(x-A)^3 - 9.51(x-A)^2 + 1.9(x-A) + 5$

9. $1 + \dfrac{x_i^2 + x_n^2}{\frac{x_i + x_n}{4} + 1}$

10. $(\frac{x}{A})^2 + (\frac{x}{B})^2 + (\frac{x}{C})^2 - 1$

(b) Each one of the following JOVIAL numeric formulas contains an error or an inefficiency. Where the meaning of the formula is clear, make the appropriate correction.

1. ((ALPHA * XX ** 2 + BETA) * XX - GAMMA)(XX + 3.)

2. ALPHA * XX + 1. / (2. * ALPHA) - 4. * ALPHA * (/XX - ALPHA/)

3. ALPHA * XX - (1. + ALPHA * XX ** 2.) / 8. * XX ** 2.

4. XX ** -2 * SIN(ALPHA) + 2 * YY ** 2 * COS(ALPHA) + 9.3124

5. ALPHA * LOG((XX + (/XX ** 2 - ALPHA ** 2/) ** .5) / * ALPHA)

6. AA(I) * (ALPHA - 2. ** XX) ** -9 + AA($I-1$) * (ALPHA - 2. ** XX) ** -6 + AA($I-2$) * (ALPHA - 2. ** XX) ** -3 + AA($I-3$)

7. (((XX - -1.) / AA($Ø$) ** 2 + (XX + -1.) / AA(1) ** 2) / (AA($Ø$) + AA(1))

8. (((ALPHA * BETA * XX + BETA * GAMMA) * XX - 9. * ALPHA * BETA * XX - 9. * GAMMA * BETA) / ((ALPHA * XX + GAMMA) * (XX + 3.) * (XX - 3.))

9. T(K) * (XX - 1.) + T($K-1$) * (XX ** 2 - 1.) + T($K-2$)

1Ø. (- (/AA/) ** -3) ** .5 + (XX ** .73468) + (/ALPHA/)

(c) Transliterate the following JOVIAL numeric formulas into the language of algebra.

1. ((XX(I) * (XX(I) ** 2 + XX(I) + 1.)) / ALPHA) ** 2

2. -1. / (XX * (XX ** 2 - 1.) ** .5)

3. - ((ALPHA / BETA ** 2) / (GAMMA ** 3 + XX ** 2)) ** XX

4. ((3 - 5 / XX + 4 / XX ** 2) / (1 + 2 / XX ** 2)) ** I

5. ((R * OMEGA * SIN(THETA)) * (R * OMEGA ** 2 * COS(THETA))
   - (R * OMEGA * COS(THETA)) * (R * OMEGA ** 2 * SIN(THETA)))
   / (R ** 2 * OMEGA ** 2 * SIN(THETA) ** 2 + R * OMEGA *
   COS(THETA) ** 2) ** .5A15

(d) The following pair of JOVIAL numeric formulas purportedly specify the same value (where XPON is a floating-point item). Choose 5 test values for XPON and evaluate both formulas to check their agreement. On the basis of these evaluations, determine which formula is likely to be more efficient.

1. 1./(((.ØØ48992 * XPON ** 2 + .Ø66512) * XPON ** 2 + 1.Ø3) * XPON
   ** 2 + 3.99416)

2. 2.71828 ** -XPON / (1. + 2.71828 ** -XPON) ** 2

(e) Given:

	Name,	Value, and	Description
of ITEM	AA	1.6384E7A-1Ø	fixed 15 Unsigned -1Ø Rounded
ITEM	BB	1.E4AØ	fixed 15 Unsigned ØØ Rounded
ITEM	CC	1.E-4A2Ø	fixed 15 Unsigned 2Ø Rounded
ITEM	XX	.333A1Ø	fixed 16 Signed 1Ø Rounded
ITEM	YY	1.E1A1Ø	fixed 16 Signed 1Ø Rounded

Estimate the values specified by the following numeric formulas, assuming that the result of an arithmetic operation is limited to fifteen magnitude bits and one sign bit. (Hint: the necessary calculations may be done in decimal, using the rule-of-thumb: ten bits is equivalent to three decimal digits.)

1. XX * YY * ((XX * 2 - YY ** 2) / (XX ** 2 + YY ** 2))

2. A / ((BB + 1.A1Ø) * (CC - 1.A1Ø))

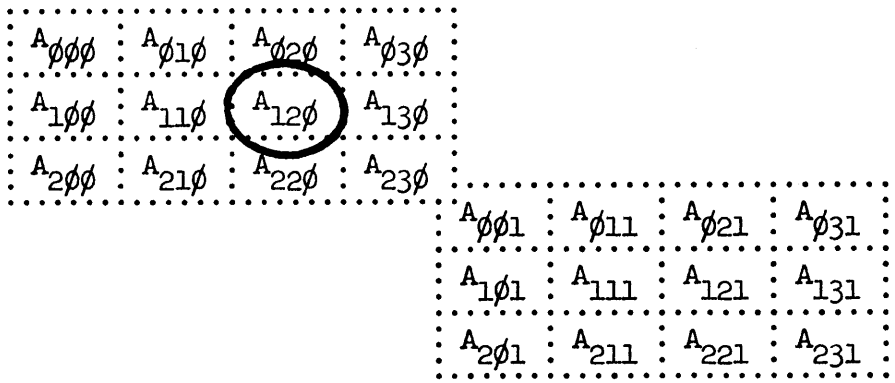
3. (AA + BB + CC) / XX / YY

4. 3.A1Ø * (1.A1Ø - XX ** 2 + BB ** 2) ** .5A1Ø

5. AA - 5.E5A1Ø * CC

INDEXES

In JOVIAL, an item name may be common to just one value, or to an entire array of values. Arrays of any dimension are possible, and 1- or 2-dimensional arrays may even be organized into tables. As in algebra, the value of an array element is designated by (item) name, subscripted by an index indicating the position of the value within the array. For example: AA($1,2,Ø$) could designate the circled value in the 3 by 4 by 2 array shown below.



A JOVIAL index is composed of a list of numeric formulas, separated by commas.

index ≠ numeric-formulas'

An index is always delimited by the subscription brackets ($ and $), and usually serves as a subscription expression following an item name. (Table names and certain functional modifiers may also be subscripted, as will be described later.)

An index is, in fact, a type of formula specifying a vector value whose components are positive integers, each of which indicates a position in the corresponding dimension of an array. Thus, an index specifying 1,2,Ø as before, indicates row 1, column 2, and plane Ø of a 3-dimensional array.

An index component may be specified by a numeric formula of any complexity. For example:

```
AA($ALPHA($I$)*I**2-BETA($I**2$),2*I,I-1$)
```

is just as correct, where subscript I is active, as AA($I,2,Ø$), and might even designate the same value. Complicated indexes of this type are useful in those cases where determining the value to be designated from an array of values requires arithmetic computation. Notice that even subscripted items may be used in indexes, leading to subscription expressions within subscription expressions -- to any level. For example: if STATE(S,I) designates the state resulting when a system in state number s is given instruction number i, then

```
STATE($STATE($STATE($STATE($STATE($Ø,A$),B$),C$),D$),E$)
```

designates the state resulting when the system, in state number Ø, is given, in sequence, instructions numbered a, b, c, d, and e.

The number of components in an index must equal the dimension of the item it subscripts. Thus, an item name designating a single value goes unsubscripted, while a linear array of values requires a 1-component index specified by a single numeric formula, and a multi-dimensional array requires a correspondingly multi-component index. Further, the value of any index component must be within the index limits of the corresponding dimension of the item being subscripted. (In JOVIAL, an n-element set has the index limits: Ø ... n-1.) For example, an index subscripting the 3 by 4 by 2 array item AA, of a previous page, must consist of exactly 3 numeric formulas, which may, in order, specify the values Ø thru 2, Ø thru 3, and Ø or 1.

Since the components of an index vector are integer values, any fractional value specified by an arithmetic formula in an index is truncated. For example: ALPHA($2/3+1/3$) is the same as ALPHA($Ø$), because Ø.666---6 + Ø.333---3 specifies Ø.999---9, which is truncated to Ø.

BASIC LITERAL CLAUSES

A literal clause describes, or designates, or specifies a literal value (i.e., a value consisting of a string of JOVIAL signs, which may be denoted by either a literal or an octal constant). Literal values are described by literal item descriptions, designated by literal variables, and specified by literal formulas.

LITERAL ITEM DESCRIPTIONS

A literal item description, composed of the Hollerith or Transmission descriptor followed by number of signs, describes a literal value in terms of the coding and size of the machine symbol representing the value.

description$_{\text{of-literal-item}}$ ≡ Hollerith;Transmission$_{\text{code}}$ number$_{\text{of-signs}}$

- The Hollerith descriptor declares the item as a literal type item whose value is represented by a Hollerith coded machine symbol. The Transmission descriptor declares the item as a literal type item whose value is represented by a Transmission-coded machine symbol. Since the JOVIAL alphabet contains 48 distinct signs, and

$2^5 = 32 < 48 < 64 = 2^6$

both coding schemes require 6 bits-per-sign for the representation of literal values.

- The number of signs after the Hollerith or Transmission descriptor declares the number of signs in the value being described and thus, indirectly, the number of bits (6 * number-of-signs) in the machine symbol representing the value.

For example, the item description

Hollerith 3

describes a 3-character literal value represented by an 18-bit, Hollerith coded machine symbol, while

Transmission 1ØØØ

describes a 1ØØØ-character literal value represented by a 6ØØØ-bit Transmission-coded machine symbol.

Hollerith coding is used in transmitting literal information to and from files (the basic input/output medium in JOVIAL) and thus varies from computer to computer. Hollerith coding is, therefore, undefined. Transmission-coding, on the other hand, is used where the literal-value/machine-symbol correspondence must be exactly defined, to ensure, for instance, a particular alphabetic order. Transmission-coding is defined on page 17.

LITERAL VARIABLES

Both simple and subscripted literal items designate literal values and are thus literal variables. For example:

```
EMPLOYEE'NAME($EMPLOYEE'NUMBER$)
FLIGHT'IDENT($F+1$)
WORD
ERROR'MESSAGE($PHASE,TYPE$)
A231($T**2/3+(/XX/)$)
```

Another type of literal variable, involving a functional modifier that allows a segment of a literal value to be designated, will be discussed later.

LITERAL FORMULAS

A literal formula specifies a literal value as denoted by a literal or octal constant, as designated by a literal variable, or as computed by a literal function.

formula$_{\text{of-literal-type}}$ ≠ literal-constant;octal-constant;variable$_{\text{of-literal-type}}$;function$_{\text{of-literal-type}}$

A literal formula thus consists of a single, literal operand.

STATUS CLAUSES

A status clause describes, or designates, or specifies a status value, which is a non-numeric value that is one of a set of qualities or categories -- as denoted by status constants, which are essentially arbitrary, though mnemonic, names. Status values are described by status item descriptions, designated by status variables, and specified by status formulas.

STATUS ITEM DESCRIPTIONS

A status item description, composed of the type descriptor Status followed by an optional number of bits and a list of status constants, describes a status value in terms of the coding and (either explicitly or implicitly) the size of the machine symbol representing the value.

description$_{\text{of-status-item}}$ ≠ Status [number$_{\text{of-bits}}$] status-constants

- The Status descriptor declares the item as a status type item.

o The optional number of bits declares the number of bits in the machine symbol representing the value of the item. When this number is omitted, machine symbol size is derived from the number of status constants, as follows:

$$2^{(\text{number-of-bits} - 1)} \leq \text{number-of-status-constants} < 2^{(\text{number-of-bits})}$$

- The list of status constants declare the possible values of the item. Any particular value of the item may thus be denoted by one of these status constants. (If a number of bits k is declared, the number of status constants should not exceed 2**k.) The order of the status constants in the description determines the coding of the item, for the sequence of values so denoted is represented by the series of integers ∅, 1, 2, 3, and so on. Consequently, no status constant may duplicate another status constant in the same item description (without making the status-constant/machine-symbol correspondence undecidable.) However, since status constants are defined only in context with the name of a particular status item, different status items may have (different) values denoted by identical status constants.

To illustrate the status item description and the concept of status values, consider:

(a) An item, QUALITY, which could be described

Status V(ROTTEN) V(BAD) V(POOR) V(MEDIOCRE) V(FAIR) V(GOOD) V(FINE) V(SUPERB)

The values of QUALITY are represented by a 3-bit machine symbol ranging numerically from ∅ for Rotten to 7 for Superb. The correspondence between the natural order of the status values, from Rotten to Superb, and the order of the machine symbol representation, from ∅ to 7, is occasionally quite useful.

(b) The item PRECEDENCE, used in ordering the sequence of message transmissions in an automated communication system. The description of PRECEDENCE is

Status V(DEFERRED) V(ROUTINE) V(PRIORITY) V(OPERATIONAL'IMMEDIATE) V(FLASH)

The values of PRECEDENCE are also represented by a 3-bit machine symbol, since there are 5 precedences and

$2^2 = 4 \leq 5 \leq 8 = 2^3$

(c) The item CARD'NUMBER, which has five possible values represented by a 5-bit positional code:

Card number	1	2	3	4	5
Binary code	ØØØØ1	ØØØ1Ø	ØØ1ØØ	Ø1ØØØ	1ØØØØ
Decimal code	1	2	4	8	16

The item, CARD'NUMBER, could be described as a status item:

Status 5 V(NULLA) V(CARD1) V(CARD2) V(NULLB) V(CARD3) V(NULLC) V(NULLD) V(NULLE) V(CARD4) V(NULLF) V(NULLG) V(NULLH) V(NULLI) V(NULLJ) V(NULLK) V(NULLL) V(CARD5)

The null statuses ensure the proper coding of the card number values and would probably not be used in the program which processes the item, CARD'NUMBER.

STATUS VARIABLES

Both simple and subscripted status items are status variables, designating status values.

STATUS FORMULAS

A status formula specifies a status value as denoted by a status constant, as designated by a status variable, or as computed by a status function.

status-formula ≠ status-constant;variable$_{\text{of-status-type}}$;function$_{\text{of-status-type}}$

BASIC BOOLEAN CLAUSES

A Boolean clause describes, or designates, or specifies a Boolean value, either True or False. Boolean values are the criteria by which a program makes logical decisions, choosing between two alternate, subsequent courses of operation. They are described by Boolean item descriptions, designated by Boolean variables, and specified by Boolean formulas.

BOOLEAN ITEM DESCRIPTIONS

A Boolean item description is composed of the Boolean descriptor, which declares the item as Boolean in type.

description$_{\text{of-boolean-item}}$ ≠ Boolean

BOOLEAN VARIABLES

Both simple and subscripted Boolean items are Boolean variables, designating either True or False. (Another type of Boolean variable involving a functional modifier will be discussed later.)

SIMPLE BOOLEAN FORMULAS

A simple Boolean formula composed of a single operand, specifies a Boolean value, True or False, as denoted by the Boolean constants 1 or $\emptyset$, as designated by a Boolean variable, or as computed by a Boolean function.

boolean-formula ≠ 1;$\emptyset$;variable$_{\text{of-boolean-type}}$;function$_{\text{of-boolean-type}}$

RELATIONAL BOOLEAN FORMULAS

A relational formula states a proposition about a relation between a pair of values, specified by arithmetic, literal, or status formulas. If, on comparison, the pair of values satisfies the relation, the relational formula specifies the value True, or if the relation is not satisfied, the formula specifies the value False. Such comparisons, as described by relational Boolean formulas, are the basis for all variable Boolean values, and thus of all programmed decisions in JOVIAL.

Relational Boolean formulas have the basic format:

formula relational-operator formula

where the formulas must be similar in type, that is: both arithmetic, both literal, or both status. (In the status case, the left-hand value must be designated by a status variable.) A relational Boolean formula involving arithmetic or literal values may contain more than one relational operator and, it follows, more than two formulas specifying

values for comparison. Each relational operator in such a portmanteau formula relates the two values specified by the arithmetic or literal formulas immediately adjacent, and the whole formula specifies True only if each relation composing it is satisfied.

boolean-formula ∉ numeric-formulas$^{\text{relational-operator}}$;literal-formulas$^{\text{relational-operator}}$;⌈variable$_{\text{of-status-type}}$ relational-operator status-formula⌉

The six relational operators signify primarily numeric relations:

Relational Operator	Conventional Notation	Meaning
EQ	=	is EQual to
GR	>	is GReater than
GQ	≥	is Greater than or eQual to
LQ	≤	is Less than or eQual to
LS	<	is LesS than
NQ	≠	is uNeQual to

For most numeric values, the significance of a relational formula is fairly obvious. For instance

I EQ 27

Either I designates the value 27 (in which case the relational formula specifies True) or it does not (and the relational formula specifies False). When a relational-formula compares fixed-point values, the precision of the comparison is never less than the precision of the least precise value, and never more than the precision of the most precise value. Unless a careful analysis has been made, therefore, it is dangerous to entrust an important program decision to an equality/inequality relation between values of widely differing precision. This, of course, also applies to floating-point values, whose precision varies.

A relational formula comparing dual fixed-point values specifies True only if both components of each value satisfy the relation. For example:

FLIGHT'POSITION(K) GR D(∅,∅)

specifies True only if flight number K's designated location is within the Northeast quadrant of the sector.

A relational formula comparing numeric values of dissimilar type invokes the implicit functions Twin, Float, and Fix. Such a comparison involving a dual fixed-point value is performed in the dual mode, and one involving a floating and a fixed value is done in the floating-point mode.

A relational Boolean formula compares non-numeric values as though the machine symbols representing them represented, instead, unsigned integers. For literal values, therefore, the significance of a relational formula depends on the ordering of the signs of the JOVIAL alphabet in the particular encoding scheme, Hollerith or Transmission code, used to represent the values. Consequently, a comparison of similarly encoded literal values determines that one value is related to the other with respect to this ordering.

In Transmission-code, the letters A thru Z and the numerals Ø thru 9 are represented by ascending sequences of numeric values, so that a relational formula involving Transmission-coded literal values can be interpreted in an alphabetic sense. This means, for example, that the relational formula 6T(ABACUS) LS 6T(ZODIAC) specifies the value True, as does 2T(99) GR 2T(ØØ).

In Hollerith code, on the other hand, both ordering and representation, though pre-determined, are undefined, so that the interpretation of relational formulas involving Hollerith-coded literal values is computer dependent.

When a relational formula compares literal values of unequal length, the shorter value is, in effect, right justified and prefixed by blanks. (In Transmission-code, blanks are represented by zero.) Thus, 4T(AAAA) GR 2T(ZZ) is entirely equivalent to 4T(AAAA) GR 4T(ZZ) in that both specify the value True. This can also be seen from the fact that 4T(AAAA) EQ 0(Ø6Ø6Ø6Ø6), and 2T(ZZ) EQ 0(3737).

Any number of relational operators may appear in a relational formula, so long as each is bracketed by a pair of arithmetic or literal formulas. The most common form, however, consists of two operators linking three formulas, the outer pair of which specify the boundaries of a range of values for the middle formula. To illustrate:

```
2**(NUMBER'OF'BITS-1) LQ NUMBER'OF'STATUS'CONSTANTS LS 2**NUMBER'OF'BITS
```

is the JOVIAL version of the formula, used previously, for determining the number of bits needed for a given number of status constants.

For status values, the significance of a relational formula again depends on the numeric encoding of the values, and thus, ultimately, on their order of declaration in the status item description. Consider, for example, the status item QUALITY, described as before:

Status V(ROTTEN) V(BAD) V(POOR) V(MEDIOCRE) V(FAIR) V(GOOD) V(FINE) V(SUPERB)

The relational formula

QUALITY(N) GR QUALITY(M)

specifies True only if "object" N is of "better" quality than "object" M. Notice that this interpretation depends on the ordering of the status constants as given in the item description, and that should this ordering change, the meaning of the formula would change with it. The relational formula

QUALITY(N) EQ V(GOOD)

on the other hand, is independent of any ordering of the status constants in the item description, and specifies True only if the value designated by the item QUALITY for "object" N is, indeed, Good.

In a relational formula involving status values, only the right-hand value may be denoted by a status or specified by a status function. And when a status is used, it must be one of those included in the description of the status item designating the left-hand value. Thus, the relational formula

QUALITY(I) NQ V(INFERIOR)

specifies neither True nor False, and is therefore meaningless and undefined, because Inferior is not one of the possible values of QUALITY. The relational formula

V(POOR) LQ QUALITY(N)

is equally incorrect, since it is ungrammatical.

COMPLEX BOOLEAN FORMULAS

A complex Boolean formula is a description of a process for computing a Boolean value, True or False, and is constructed of simple and relational Boolean formulas connected by logical operators. The value of such a formula depends both on its logical structure and on the values of the less complex formulas composing it.

boolean-formula ≠ ⌈(boolean-formula)⌉ ; ⌈⌊NOT⌋ boolean-formulas ⌈AND ; OR⌉⌉

The grammar of complex Boolean formulas is analogous to that of numeric formulas in containing brackets, and both unary and binary operators. The parentheses group Boolean sub-formulas in the ordinary manner. The unary operator NOT reverses the value specified by the Boolean formula immediately following it from True to False or from False to True. The value of any Boolean formula may therefore be reversed by preceding it with the logical operator NOT. For example:

NOT QUALITY(R) EQ V(FAIR)

The binary operator AND combines the two Boolean values specified by the formulas on either side of it to yield the value True only if both operand formulas specify True. The binary operator OR also combines the values specified by the adjacent Boolean formulas and yields the value False only if both operand formulas specify False. The following examples illustrate the logical operators AND and OR in use:

QUALITY(R) EQ V(FAIR) AND R NQ 27

QUALITY(R) EQ V(FAIR) OR QUALITY(R) EQ V(FINE)

The meaning of the logical operators is summarized in the following chart, where p and q signify Boolean values:

If	p	is	False	False	True	True
and	q	is	False	True	False	True
then	NOT q	is	True	False	True	False
	p AND q	is	False	False	False	True
	p OR q	is	False	True	True	True

Unless parenthese indicate otherwise, the precedence of the logical operations is: NOTs first; ANDs second; ORs last. Within these three categories, operations are performed from left to right, and the formula's value is completely specified as soon as the evaluation of a sub-formula conclusively determines a value for the entire formula. For example: in a set of Boolean sub-formulas connected by ANDs, the value False is specified for the entire formula as soon as any sub-formula specifying False is evaluated; and similarly, a set of Boolean sub-formulas connected by ORs specifies True as soon as any sub-formula specifying True is evaluated. In such formulas, execution time can be saved by placing toward the left those sub-formulas most likely to be: False in the case of connecting ANDs; and True in the case of connecting ORs.

It should be noted that the Boolean values True and False, along with the three logical operators NOT, AND, and OR, constitute what is known as a Boolean algebra. A complex Boolean formula may therefore be manipulated according to the rules of Boolean algebra in much the same way that an arithmetic formula is manipulated according to the rules of ordinary algebra. The object of such manipulation on the part of the JOVIAL programmer is twofold: (1) to simplify the formula; and (2) to uncover tautologies and contradictions.

A tautology is a Boolean formula that always specifies True no matter what the truth-value of its component simple and relational Boolean formulas.

A contradiction is a Boolean formula that always specifies False no matter what the truth-value of its component simple and relational Boolean formulas.

Tautologies and contradictions may always be removed from any program in which they occur, for logical decisions must be based on Boolean formulas which are neither tautologies nor contradictions to really choose between alternate paths of action.

A Boolean formula that is a tautology can always be reduced, by application of the theorems and axioms of Boolean algebra, given below, to the form of one of the following two basic tautologies:

1. p OR NOT p

2. p OR NOT p OR q

It is easy to see that no matter what truth-values are assigned to p and q, the above formulas are always True. Contradictions can similarly be reduced to a negation of one of these forms.

For readers unfamiliar with Boolean algebra, a list of its most useful axioms and theorems follows. The form following the ≢ may be substituted for the form preceding (and vice-versa) without affecting truth-values.

1. NOT (NOT p) ≢ p
2. p AND p ≢ p
3. p OR p ≢ p
4. p AND (q OR NOT q) ≢ p
5. p OR (q AND NOT q) ≢ p
6. p AND (p OR q) ≢ p
7. p OR (p AND q) ≢ p
8. p AND q ≢ q AND p

9.	p OR q	≢	q OR p
10.	p AND (q AND r)	≢	(p AND q) AND r
11.	p OR (q OR r)	≢	(p OR q) OR r
12.	(p AND q) OR (p AND r)	≢	p AND (q OR r)
13.	(p OR q) AND (p OR r)	≢	p OR (q AND r)
14.	NOT (p AND q)	≢	NOT p OR NOT q
15.	NOT (p OR q)	≢	NOT p AND NOT q

In addition to the above equivalences based on the properties of the logical operators, the following equivalences based on the properties of the relational operators may prove useful, where x and y are a pair of arithmetic, literal, or status formulas.

16.	NOT (x EQ y)	≢	x NQ y	≢	y NQ x
17.	NOT (x GR y)	≢	x LQ y	≢	y GQ x
18.	NOT (x GQ y)	≢	x LS y	≢	y GR x
19.	NOT (x LQ y)	≢	x GR y	≢	y LS x
20.	NOT (x LS y)	≢	x GQ y	≢	y LQ x
21.	NOT (x NQ y)	≢	x EQ y	≢	y EQ x

To illustrate some of the techniques of working with Boolean formulas, consider the following items and their descriptions:

```
ITEM       PRICE fixed 15 Unsigned $
ITEM     QUALITY Status V(ROTTEN) V(BAD) V(POOR) V(MEDIOCRE) V(FAIR)
                 V(GOOD) V(FINE) V(SUPERB) $
ITEM REQUIREMENT Boolean $
```

and the process of simplifying the following complex Boolean formula involving these items.

```
((1ØØ LQ PRICE($I$) LQ 1ØØØØ AND QUALITY($I$) EQ V(BAD) AND REQUIREMENT($I$))
OR ((1ØØ LQ PRICE($I$) LQ 1ØØØØ AND QUALITY($I$) EQ V(BAD) AND NOT
REQUIREMENT($I$)) OR ((1ØØ LQ PRICE($I$) LQ 1ØØØØ AND QUALITY($I$) EQ
V(ROTTEN)))
```

The first step is to isolate the simple and relational Boolean formulas so that they may be replaced by even simpler symbols to better show the logical structure of the formula:

(1ØØ LQ PRICE(I) LQ 1ØØØØ) ≡ p

(QUALITY(I) EQ V(BAD)) ≡ q_1

REQUIREMENT(I) ≡ r

(QUALITY(I) EQ V(ROTTEN) ≡ q_2

The second step is to rewrite the formula using the artificial symbols just generated:

(p AND q_1 AND r) OR (p AND q_1 AND NOT r) OR (p AND q_2)

and then, simplify the rewritten formula using the rules of Boolean algebra:

≡ p AND ((q_1 AND r) OR (q_1 AND NOT r) OR q_2) by #12

≡ p AND (q_1 AND (r OR NOT r) OR q_2) by #12

≡ p AND (q_1 OR q_2) by #4

Returning to JOVIAL, the simplified formula becomes:

(1ØØ LQ PRICE(I) LQ 1ØØØØ) AND (QUALITY(I) EQ V(BAD) OR QUALITY(I) EQ V(ROTTEN))

Due to the structure of the status item quality, a still further simplification could be effected:

(1ØØ LQ PRICE(I) LQ 1ØØØØ) AND (QUALITY(I) LQ V(BAD))

EXERCISE (Boolean Formulas)

(a) Using the items price, quality, and requirement as described in the last example, simplify the following Boolean formulas, checking for tautologies and contradictions.

1. (PRICE(E) GR 65ØØØ AND QUALITY(E) EQ V(FINE)) OR REQUIREMENT(E) AND PRICE(E) GR 65ØØØ)

2. PRICE(E) LQ 999 OR QUALITY(E) EQ V(FAIR) OR NOT PRICE(E) LS 1ØØØ

3. NOT (PRICE(E) LQ PRICE($E-1$) AND QUALITY(E) NQ QUALITY($E-1$))

4. NOT PRICE(E) LS PRICE($E-1$) AND PRICE($E-1$) LQ PRICE(E)

5. QUALITY(E) EQ V(MEDIOCRE) AND (PRICE(E) EQ 65ØØØ OR (REQUIREMENT(E) AND QUALITY(E) EQ V(SUPERB) AND PRICE(E) EQ Ø))

6. (PRICE(E)**2 LQ 49ØØ OR QUALITY(E) EQ V(SUPERB) OR NOT REQUIREMENT(E)) AND (PRICE(E)**2 LQ 49ØØ OR QUALITY(E) EQ V(SUPERB) OR REQUIREMENT(E)) AND (PRICE(E) LQ 5Ø)

7. ((QUALITY(E) EQ V(SUPERB)) OR ((QUALITY(E) EQ V(SUPERB)) AND (REQUIREMENT(E))) OR ((PRICE(E) LQ 65ØØØ) OR ((QUALITY(E) EQ V(SUPERB)) AND (PRICE(E) LQ 65ØØØ))))

8. NOT ((REQUIREMENT(E)) OR NOT (REQUIREMENT(E)) AND (PRICE(E) LS 1ØØØ))

9. (PRICE(E) LS 1ØØØ AND NOT REQUIREMENT(E) AND QUALITY(E) NQ V(ROTTEN)) OR (PRICE(E) LS 1ØØØ AND QUALITY(E) NQ V(ROTTEN) AND NOT REQUIREMENT(E)) OR NOT REQUIREMENT(E)

10. (NOT (QUALITY(E) EQ V(FINE)) OR ((NOT REQUIREMENT(E)) AND (REQUIREMENT(E)))) OR (((PRICE(E) LS 5Ø) AND (PRICE(E) GQ 35)) OR NOT (QUALITY(E) EQ V(FINE))) OR (NOT (QUALITY(E) EQ V(FINE))) OR ((PRICE(E) LS 5Ø) AND (PRICE(E) GQ 35))

SENTENCES

With certain delimiters, clauses are combined to form statements and declarations, which are the sentences of JOVIAL. Statements assert actions that the program is to perform, and declarations describe the environment in which the actions are to occur.

The remainder of this document deals with the construction of such sentences and with their combination to form programs and procedures.

BASIC DATA DECLARATIONS

A major part of the environment of any program consists of the data it must manipulate. In machine-oriented programming, data description is buried within the program, as machine instructions for accessing the data each time it is required. In JOVIAL, data is described once, with a set of data declarations, and is thereafter referred to by name. When the data changes slightly, only these declarations need be modified and, upon re-compilation, all the necessary changes are reflected in the resulting machine language program.

Data declarations have no operational meaning, and their descriptive affect is not alterable by the execution of the program.

They merely determine how the program is to manipulate the data values in its environment, making it necessary to precede any reference to a data value with a defining declaration. These do not necessarily have to be supplied by the programmer but may appear in a COMPOOL.

ITEM DECLARATIONS. In data processing, the natural unit of information is the value. In JOVIAL, values are denoted by constants, designated by variables, and specified by formulas. Values other than those denoted by constants or utilized only as intermediate results (e.g., in a Boolean formula) must be formally declared as items.

An item declaration, composed of the declarator ITEM followed by a name and an item description, and terminated by the $ separator, declares the identifier to be an item name designating a single value of the type given in the item description.

declaration ≠ ITEM name$_{\text{of-item}}$ description $

- The ITEM declarator begins the declaration.
- The identifier is declared to be a simple item name, designating the item's value.
- The floating, fixed, dual, literal, status, or Boolean item description declares the item's type and describes its value in terms of the machine symbol representing it.
- The $ separator terminates the declaration.

The following is a summary of JOVIAL item declarations. For more detailed explanation, refer to the sections on the various item descriptions.

ITEM name Floating ⌊Rounded⌋ ⌊floating-constant ... floating-constant⌋ $

ITEM name fixed number$_{\text{of-bits}}$ Signed;Unsigned ⌊number$_{\text{of-fraction-bits}}$⌋
⌊Rounded⌋ ⌊integer;fixed-constant ... integer;fixed-constant⌋ $

ITEM name Dual number$_{\text{of-bits}}$ Signed;Unsigned ⌊number$_{\text{of-fraction-bits}}$⌋
⌊Rounded⌋ ⌊dual-constant ... dual-constant⌋ $

ITEM name Hollerith;Transmission$_{code}$ number$_{of\text{-}signs}$ $

ITEM name Status ⌊number$_{of\text{-}bits}$⌋ status-constants $

ITEM name Boolean $

Examples of the six types of item declaration are given below.

ITEM P6 Floating Rounded ∅.1E-1∅...∅.1E+1∅ $

declares the name P6 to be an item name designating a floating-point value that was rounded upon being assigned to the item and that is expected to range in magnitude from .∅∅∅∅∅∅∅∅∅1 to 1∅∅∅∅∅∅∅∅∅.

ITEM TALLY fixed 15 Unsigned $

declares the name TALLY to be an item name designating a fixed-point, 15-bit, unsigned, integral value.

ITEM PLACE Dual 16 Signed 5 $

declares the name PLACE to be an item designating a dual fixed-point, 16-bit-per-component, signed value with 5 fraction bits per component.

ITEM IDENT Hollerith 12 $

declares the name IDENT to be an item name designating a Hollerith coded, 12-character, literal value.

ITEM OP'TYPE Status 6 V(ARITH) V(RELAT) V(LOGIC) V(OTHER) $

declares the name OP'TYPE to be an item name designating a status value, encoded in 6 bits rather than the minimal 3, signifying Arith, Relat, Logic, or Other.

ITEM SPARE'INDICATION Boolean $

declares the name SPARE'INDICATION to be an item name designating a Boolean value.

MODE DECLARATIONS. When a program requires that many individual values of similar type be given explicit definition, it is somewhat tedious to have to write an entire set of item declarations, identical but for name.

For example: a sizeable program for reducing floating-point test data might require scores of simple, floating-point items just for temporary results. The mode declaration eliminates the necessity of declaring each of these items separately.

A mode declaration, composed of the MODE declarator followed by an item description and terminated by the $ separator, declares an implicit mode of definition for subsequent simple item names that are otherwise undefined.

declaration ≠ MODE description $

- The MODE declarator begins the declaration.

- The item description defines simple and otherwise undefined item names, listed after the declaration, by declaring their type and describing their value.

- The $ separator terminates the declaration.

The effect of a mode declaration depends on its place in the JOVIAL program listing since it can only apply to subsequently used item-names. After a mode declaration, the initial occurrence of any unsubscripted and undefined label, in any context where an item name is expected, serves at that point to declare the label an item name, defined according to the mode. If simple items with different definitions are also needed, they may, of course, be explicitly declared (prior to their initial use, however, lest they too be defined "a' la mode").

A mode declaration remains effective until it is superseded by another mode declaration. This means that different portions of a program may have their own, local modes of implicit item definition. Thus, one part of a JOVIAL program might deal mostly with floating-point values, so that

MODE Floating Rounded $

would be applicable. Another portion might require that integers be the mode, and

MODE fixed 16 Signed $

would be used. For programming vector calculations,

MODE Dual 24 Signed 1Ø $

might be convenient, and a procedure for manipulating 5-character literal values might be easier to write if

MODE Transmission 5 $

were used. A routine involving many similar status items might be preceded by

MODE Status V(STATEØ) V(STATE1) V(STATE2) V(ETC) $

and a logically complex routine might need

MODE Boolean $

ARRAY DECLARATIONS. An array declaration describes the structure of a collection of similar values, and also provides a means of identifying this collection with a single item name. In JOVIAL, therefore, an array is a structured collection of similar values identified by a single item name. A vector is an example of a one-dimensional or linear array, and a matrix is an example of a two-dimensional array. Rectangular arrays of any dimension may be declared, but one and two dimensional arrays are by far the most common.

An array declaration, composed of the ARRAY declarator followed by a name, a list of dimension numbers, an item description, and terminated by the $ separator, declares the name to be an array item name designating an array of values with the structure imposed by the dimension numbers and of the type given in the item description.

declaration ≠ ARRAY name$_{\text{of-array-item}}$ numbers$_{\text{of-elements-per-dimension}}$ description $

- The ARRAY declarator begins the declaration.

- The identifier is declared to be an array item name, designating all the array's values. Since each value in the array bears this name, individual values must be distinguished by an index subscripting the name.

- A dimension number declares the size of one dimension of the array: first, the number of rows; second, the number of columns; third, the number of planes; and so on.

- The item description declares the array's type and describes its values in terms of the machine symbols representing them.

- The $ separator terminates the declaration.

In designating an individual value from an n-dimensional array, the array item name must be subscripted by an n-component index. And where the size of a dimension is k, the value of the corresponding component of the index can only range from ∅ thru k-1.* To illustrate, consider the declaration

ARRAY ALPHA 2 4 3 fixed 7 Unsigned ∅...99 $

which declares ALPHA a 2 by 4 by 3 array of positive integers less than 1∅∅. A typical set of values for ALPHA is:

89	13	21	48
46	14	83	12
17	92	99	65
57	74	∅5	22
69	34	32	17
∅∅	1∅	36	15

Thus, ALPHA($∅,2,1$) specifies 99, and ALPHA($1,3,2$) specifies 15. The index 1,3,2 is the maximum index that may subscript ALPHA.

To further illustrate arrays and array declarations, consider:

(a) The declaration

ARRAY IDENT 6 Transmission 8 $

which declares IDENT a 6-element vector of 8-character, Transmission-coded, literal values. A typical set of values for IDENT is:

8T(UNKNOWN)
8T(PENDING)
8T(HOSTILE)
8T(FRIENDLY)
8T(FRIENDLY)
8T(PENDING)

* This means that the first element of any array is indexed by zeros. Thus, MATRIX($∅,∅$) is the first element of a 2-dimensional array.

and IDENT(2) specifies 8T(HOSTILE).

(b) The declaration

```
ARRAY TIC'TAC'TOE  3 3  Status V(EMPTY) V(NOUGHT) V(CROSS) $
```

which declares TIC'TAC'TOE a 3 by 3 matrix of rather obvious status values. A typical set of values for TIC'TAC'TOE is:

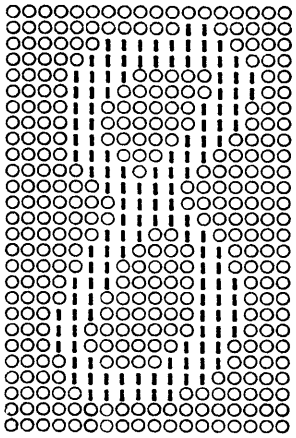
V(CROSS)	V(NOUGHT)	V(EMPTY)
V(NOUGHT)	V(EMPTY)	V(EMPTY)
V(CROSS)	V(EMPTY)	V(CROSS)

and whether NOUGHT's plays TIC'TAC'TOE($1,1$) or TIC'TAC'TOE($2,1$), CROSS's will win.

(c) The declaration

```
ARRAY CHARACTER 27 18 Boolean $
```

which declares a 27 by 18 Boolean matrix. A typical set of values for CHARACTER is:



and much current research effort is devoted to the development of general processing rules for converting such a matrix into the value 1H(8).

BASIC STATEMENTS

The programmer's job, once the problem has been analyzed and the data described, is to determine a sequence of processing operations which will transform the input data into a solution for the problem. In JOVIAL, the fundamental unit of expression in the description of processing operations is the statement. Thus, a sequence of processing operations may be described by a list of JOVIAL statements. (Such a list, interspersed with declarations describing the data, forms a program which may be transformed by a JOVIAL compiler into an equivalent, machine-(oriented)-language program.)

JOVIAL statements are normally executed* in the sequence in which they are listed. Few problems, however, are so simple that a single, unbroken sequence of processing operations will suffice to provide solutions from all possible sets of input data. Consequently, the programmer must provide alternate sequences, and determine processing operations which allow the computer to choose between them. Simple JOVIAL statements are therefore divided into two classes: (1) statements which affect the data, by varying variable values; and (2) statements which affect (the execution of) the program, by altering the normal sequence of statement execution.

NAMED STATEMENTS. It is often necessary to attach a name to a statement so that it may be referred to elsewhere in the program and executed out of its normal, listed sequence. A named statement, composed of a statement name followed by the . separator and a statement, may be referenced for this purpose at any point in the program.

statement ≡ name$_{\text{of-statement}}$. statement

Any JOVIAL statement -- simple, compound, or even already named -- may be identified in this manner, but a statement name is needed only when the statement is to be executed out of sequence.

A statement name may describe the purpose of the statement, as in

COMPUTE'TAX. Statement

or it may merely indicate a relative position in the program listing, as in

---

* By "the execution of a statement" is meant the computer's execution of the machine-language routine compiled from the statement.

STEP'19. Statement

COMPOUND STATEMENTS. It is frequently desirable to group several statements together into a larger form which itself is to be considered a single statement. Such a statement is called a compound statement, and is composed of the BEGIN bracket followed by a list of statements (possibly interspersed with declarations) and terminated by the END bracket.

statement ↞ BEGIN ⌈declaration;statement⌉s END

A compound statement is completely equivalent to a single, simple statement. The BEGIN and END brackets serve as opening and closing statement parentheses and, since the statements they enclose may themselves be compound, whole strings of BEGINs or, more commonly, of ENDs often occur. For example:

```
BEGIN
Statement
     BEGIN
     Statement
          BEGIN
          Statement
          Statement
          END
     END
END
```

The indentation is a useful typographical device, helping to match BEGINs and ENDs. The correct matching of BEGINs and ENDs is as important to the meaning of a program as the correct matching of parentheses is to the meaning of a numeric formula.

Certain other forms, involving the grouping of several statements, automatically constitute compound statements and will be discussed in greater detail later.

ASSIGNMENT STATEMENTS. An assignment statement, composed of a variable followed by the = separator and a formula, and terminated by the $ separator, assigns the value specified by the formula to be the value thereafter designated by the variable.

statement ↞ variable = formula $

This is accomplished by evaluating the formula, and then replacing the machine symbol representing the value of the variable with the result. The formula must, however, specify a value of the same type as the variable, that is, formula and variable must both be: numeric; literal; status; or Boolean.

As used in an assignment statement, the = separator signifies the process of replacement. Thus:

```
OMEGA = GAMMA*SIN(ALPHA-BETA) $
```

means, "the value designated by OMEGA is replaced by the value specified by GAMMA*SIN(ALPHA-BETA)." An assignment statement is therefore, in effect, a command to compute and to substitute, and should not be confused with an algebraic equation where the equals sign has the meaning of the JOVIAL relational operator EQ. For example, the construction

```
2*X = Y+2 $
```

has no meaning in JOVIAL, while

```
X = (Y+2)/2 $
```

is a valid statement that can be executed by a compiled program. The distinction is best appreciated when assignment statements such as

```
ALPHA = ALPHA + 27. $
```

are considered. This perfectly legal assignment statement merely describes the process of incrementing the value of the item ALPHA by twenty-seven, and does not claim that zero and twenty-seven are equal.

NUMERIC ASSIGNMENT STATEMENTS. In a numeric assignment statement, both variable and formula are numeric in type. Numeric values, however, have three modes of representation: floating-point; fixed-point; and dual fixed-point, and a numeric assignment statement may mix any two of these modes. When this occurs, the implicit conversion functions Fix, Float, and Twin are necessarily invoked. The following cases must be considered:

(a) A floating-point value may be assigned to any numeric variable. Assignment to a fixed-point variable implies conversion to fixed-point representation. Assignment to a dual fixed-point variable implies: first, a conversion to fixed-point; and then, a twinning conversion to dual.

(b) A fixed-point value may also be assigned to any numeric variable. Assignment to a floating-point variable implies conversion to floating-point representation. Assignment to a dual fixed-point variable implies a twinning conversion to dual.

(c) A dual fixed-point value may only be assigned to a dual variable, since no implicit "Untwin" conversion exists.

(d) A negative value, in any mode of representation, may only be assigned to a signed variable. Since floating-point variables are implicitly signed, this restriction only applies to unsigned fixed-point and dual fixed-point variables.

When a numeric value is assigned to a fixed-point variable, the machine-symbol representing the value must be tailored to fit the format of the variable, both in significance and in precision. Thus, if the specified value has less significance than the variable, its machine symbol representation is prefixed by leading zeros upon assignment, or if the specified value is less precise than the variable, its machine symbol representation is suffixed by trailing zeros. If, on the other hand, the specified value has more significance than the variable, this excess significance may be lost upon assignment (a condition known as "overflow"), or if the specified value is more precise than the variable, excess precision is lost, either by truncation or by rounding. To illustrate these concepts, consider:

```
ITEM ALPHA fixed Ø5 Signed 1 $  ''Format  s     o o i i i.f o o    ''
ITEM  BETA fixed Ø9 Signed 3 $  ''Format  s     i i i i i.f f f    ''
ITEM GAMMA fixed 13 Signed 5 $  ''Format  s i i i i i i i.f f f f f''
```

and the assignment statements:

```
BETA = ALPHA $
BETA = GAMMA $
```

The effect of these two assignment statements can be readily seen by comparing the machine symbol formats above. Upon assignment to BETA, the value of ALPHA has both leading and trailing zero magnitude bits affixed, while the value of GAMMA loses both precision, as underlined, and probably significance, as doubly underlined.

Unforeseen loss of significance, or overflow, occurs in fixed-point assignments as the result of a programming error: failure to declare enough bits for the variable item. In the case of the assignment

```
BETA = GAMMA $
```

the error is obvious, and is easily corrected by revising the item

declaration for BETA so that seven rather than five integer bits are declared. Other cases, however, are not quite so obvious and a detailed analysis may be necessary.* Consider

```
BETA = (/31*ALPHA+GAMMA/)**Ø.5A3+13.A3 $
```

By substituting values of ALPHA and GAMMA which maximize the value specified by the formula (i.e., +7.5A1 for ALPHA, and +127.97A5 for GAMMA), the maximum possible value assigned to BETA can be determined. This value is 31.99, and would be truncated to 31.875 without overflow upon assignment to BETA. Had BETA been declared rounded, though, as in

```
ITEM BETA fixed Ø9 Signed 3 Rounded $
```

the rounding of 31.99 to 32.+ by the effective addition of 2^{-4} (or .Ø625) during assignment, could create an overflow condition, since 31.875 is the maximum value that BETA can designate.

In this paragraph, some examples of numeric assignment statements are presented as solutions to the following problem: As the result of previous computation, values a and c have been assigned to the floating-point items AA and CC. A JOVIAL statement is needed to assign x and y as the values of the floating-point items XX and YY, where:

$$x = \frac{1}{\frac{1}{a} + \frac{1}{c}}, \qquad y = \sqrt[3]{\sqrt{\left(\left(\frac{a}{3}\right)^3 + \frac{c}{2}\right)^2 - \left(\frac{a}{3}\right)^6} - \left(\left(\frac{a}{3}\right)^3 + \frac{c}{2}\right)}$$

$$- \sqrt[3]{\sqrt{\left(\left(\frac{a}{3}\right)^3 + \frac{c}{2}\right)^2 - \left(\frac{a}{3}\right)^6} + \left(\left(\frac{a}{3}\right)^3 + \frac{c}{2}\right)} - \frac{a}{3}$$

```
SOLUTION1.  BEGIN
            XX = 1./(1./AA+1./CC) $
            YY = ((((AA/3.)**3+CC/2.)**2-(AA/3.)**6)**.5
-((AA/3.)**3+CC/2.))**.3333333333 - ((((AA/3.)**3+CC/2.)**2-(AA/3.)
**6)**.5+((AA/3.)**3+CC/2.))**.3333333333 - AA/3. $
            END
```

Notice that the formula specifying the value y contains duplicate sub-formulas. To avoid inefficiently computing the same value twice, the

---

* Where such an analysis is either difficult or impossible, floating-point representation should be considered.

computation can be broken down into several smaller assignment statements, saving such intermediate results for later use. If the values a and c are not needed for further computation, the items AA and CC can be used, along with YY, for temporary storage, thus eliminating a need for the declaration of two extra floating-point items for this purpose.

```
SOLUTION2.  BEGIN
            XX = 1./(1./AA+1./CC) $
            AA = AA/3. $
            YY = AA**3 $
            CC = YY+CC/2. $
            YY = (CC**2-YY**2)**.5 $
            YY = (YY-CC)**.3333333333-(YY+CC)**.3333333333-AA $
            END
```

LITERAL ASSIGNMENT STATEMENTS. In a literal assignment statement, both variable and formula are literal in type. Where the value being assigned and the variable being set are similarly encoded (i.e., both Hollerith or both Transmission-code) and equal in size, the effect of a literal assignment statement is clear. Thus, given

```
ITEM C'DATA Transmission 6 $
```

the statement

```
S1.   C'DATA = 6T( JOINT) $
```

cannot be misinterpreted. If, however, the literal value being assigned were larger than the variable, for example:

```
S2.  C'DATA = 1ØT(CLIP JOINT) $
```

then its leading characters would be lost ("CLIP" in the above case). And, finally, if the literal value being assigned were smaller than the variable, as in

```
S3.  C'DATA = 5T(JOINT) $
```

then it would be right justified and prefixed by the necessary number of blank characters. Notice that the three literal assignment statements, S1, S2, and S3, all produce the same effect on the literal variable C'DATA.

Where the value being assigned and the value being set are not similarly encoded (i.e., one is Hollerith, the other, Transmission-code), the above considerations on size pertain, but the results are, in general, undefined -- to the same extent that Hollerith representation itself is undefined. (Any prefixed blanks are encoded according to the representation of the variable being set.)

STATUS ASSIGNMENT STATEMENTS. In a status assignment statement, the value assigned to the status item variable may be denoted by one of its status constants, designated by another status item, or computed by a status function. Where the specified value is denoted by a status constant, it must, of course, be one of the declared status constants of the item being assigned, for example:

```
QUALITY($K$) = V(BAD) $
```

And where the specified value is designated by a status item or computed by a status function, a meaningful correspondence should exist between the set of status values that may be specified by the formula, and the set of status values that may be designated by the variable item. When these are identical sets of similarly encoded status values, the correspondence is obvious. Thus

```
QUALITY($K$) = QUALITY($I$) $
```

states that value K of QUALITY is to be the same as value I of QUALITY. Since an assignment statement manipulates status values as integers, any correspondence between different sets of status values is based on their numerical encoding, and not on any coincidental similarities in the status constants themselves.

BOOLEAN ASSIGNMENT STATEMENTS. In a Boolean assignment statement, both variable and formula are Boolean in type. A Boolean assignment statement, therefore, assigns a Boolean variable that value, True denoted by 1 or False denoted by $\emptyset$, specified by a Boolean formula.

Some examples of Boolean assignment statements are presented in the solution to the following problem: The environment contains two 27 by 18 Boolean arrays, GRID1 and GRID2. Previous statements have activated the subscripts I and J, and assigned them values. Write a JOVIAL statement assigning the value True to item I,J of GRID2 if the corresponding value of GRID1 is either True or is bracketed (horizontally, vertically, or diagonally) by a pair of true values, and assigning the value False otherwise. (The iteration of this statement over an entire array would fill in small holes in the pattern.) Notice: There are, obviously, only four possible ways a GRID1 value may be bracketed by a pair of True's,

```
010  000  001  100
0 0  1 1  0 0  0 0
010  000  100  001
```

and these are all applicable only when the center value is, indeed, an interior value of the array.

```
          SOLUTION.  BEGIN
MODE                 Boolean $
                     INSIDE'ROW = ∅ LS I LS 26 $
                     INSIDE'COLUMN = ∅ LS J LS 17 $
                     CASE1 = INSIDE'ROW AND GRID1($I-1,J$) AND
GRID1($I+1,J$) $     CASE2 = INSIDE'COLUMN AND GRID1($I,J-1$) AND
GRID1($I,J+1$) $     CASE3 = INSIDE'ROW AND INSIDE'COLUMN AND
GRID1($I-1,J-1$) AND GRID1($I+1,J+1$) $
                     CASE4 = INSIDE'ROW AND INSIDE'COLUMN AND
GRID1($I-1,J+1$) AND GRID1($I+1,J-1$)$
                     GRID2($I,J$) = GRID1($I,J$) OR CASE1 OR CASE2
OR CASE3 OR CASE4 $  END
```

The Boolean mode declaration allows temporary Boolean items to be used without explicitly declaring them. These items are not strictly necessary since the entire function of the above statement could be compressed into a single assignment, but the resulting Boolean formula would be both large and computationally inefficient due to duplicated sub-formulas. The greatest computational efficiency, however, could probably be attained by combining the last five assignment statements into one, to take advantage of the fact that all Boolean sub-formulas need not always be evaluated for the entire formula to specify a value.

EXCHANGE STATEMENTS. An exchange statement, composed of a variable followed by the == separator and another variable, and terminated by the $ separator, exchanges the values designated by the two variables.

```
statement  ϕ  variable == variable $
```

The effect of an exchange statement on either of the variables involved is as if each had been assigned the value designated by the other. Consequently, the rules of assignment pertain, and both variables must be the same type: numeric; literal; status; or Boolean.

An exchange statement is operationally equivalent to a sequence of three assignment statements -- augmented by a declaration of temporary storage. To illustrate, consider the exchange statement

```
AA == BB  $
```

which has precisely the effect of

```
                BEGIN
ITEM TEMPORARY ''Include here an item description duplicating that of
                item AA.'' $
                TEMPORARY = AA $
                AA = BB $
                BB = TEMPORARY $
                END
```

except, of course, that an exchange statement does not explicitly declare an item named TEMPORARY.

To emphasize the effect of the exchange statement, consider the following rather roundabout way of assigning the value True to BOOL, a Boolean variable.

```
                BEGIN
ITEM      ALPHA Transmission 5 $
ITEM      OMEGA Transmission 5 $
                ALPHA = 5T(ALPHA) $
                OMEGA = 5T(OMEGA) $
     EXCHANGE. ALPHA == OMEGA $
                BOOL = ALPHA EQ 5T(OMEGA) AND OMEGA EQ 5T(ALPHA) $
                END
```

IF STATEMENTS. It often happens during a calculation that the computer must choose between one of two alternate sequences of operation depending on whether a specified criterion is or is not met. A good example is the calculation of gross pay for an hourly employee who gets time-and-a-half for hours worked in excess of 4Ø per week. Clearly, the basis for choice is: Has the man worked more than 4Ø hours this week? For employee number E, this question can be expressed as a Boolean formula in an IF statement:

```
IF HOURS'WORKED($E$) GR 4Ø $
```

Where a criterion is expressed as a Boolean formula, an IF statement serves to choose between two alternate sequences of operation. An IF statement, composed of the IF sequential operator followed by a Boolean formula and terminated by the $ separator, causes the next statement listed to be executed or skipped depending on whether the Boolean formula specifies True or False.

statement ≙ IF boolean-formula $

In other words, if the Boolean formula of the IF statement specifies the value True, the statement following it is executed. If, on the

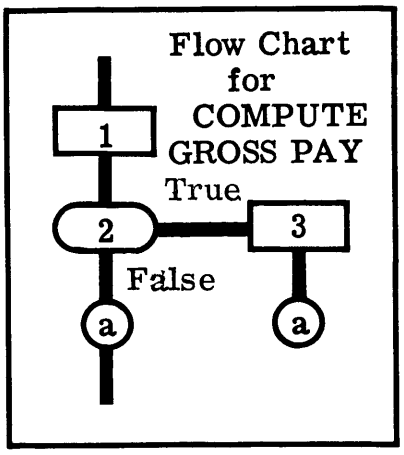
other hand, the Boolean formula of the IF statement specifies the value False, the statement following it is skipped, and operation continues with the next statement listed.

A JOVIAL statement computing gross pay for employee E can therefore be written as

```
       BEGIN COMPUTE'GROSS'PAY.
STEP1. GROSS'PAY($E$) = HOURS'WORKED($E$)*HOURLY'PAY($E$) $
STEP2. IF HOURS'WORKED($E$) GR 40 $
STEP3.      GROSS'PAY($E$) = GROSS'PAY($E$)+(HOURS'WORKED($E$)-40)
* HOURLY'PAY($E$) / 2 $
       END
```

In STEP1, the employee's gross pay is computed at straight-time for all hours worked. In STEP2, if the employee has worked more than 40 hours, then, in STEP3, his gross pay is incremented at half-time for the excess hours, thus paying time-and-a-half for overtime.

The statement execution flow for this process can be graphically illustrated, as in the accompanying diagram.



It is important to realize that the conditional statement following an IF statement can be compound as well as simple. For example:

```
STEP1. IF GAMMA LS DELTA $
            BEGIN
STEP2.      ALPHA = LOG(GAMMA)
                  + DELTA**2 $
STEP3.      BETA  = EXP(GAMMA)
                  + ALPHA**3 $
            END
STEP4. IF GAMMA EQ DELTA $
            BEGIN
STEP5.      ALPHA = 0. $
STEP6.      BETA  = 0. $
            END
STEP7. IF GAMMA GR DELTA $
            BEGIN
STEP8.      ALPHA = LOG(DELTA)
                  - GAMMA**2
STEP9.      BETA  = GAMMA**.5 $
            END
```

One and only one of the three compound statements above will be executed, although this is not apparent from the statement execution flow.

The use of the IF statement in decision-making will be illustrated by some further examples, wherein the environment contains SCORE, a linear, 4-element floating-point array declared

```
ARRAY SCORE 4 Floating Rounded $
```

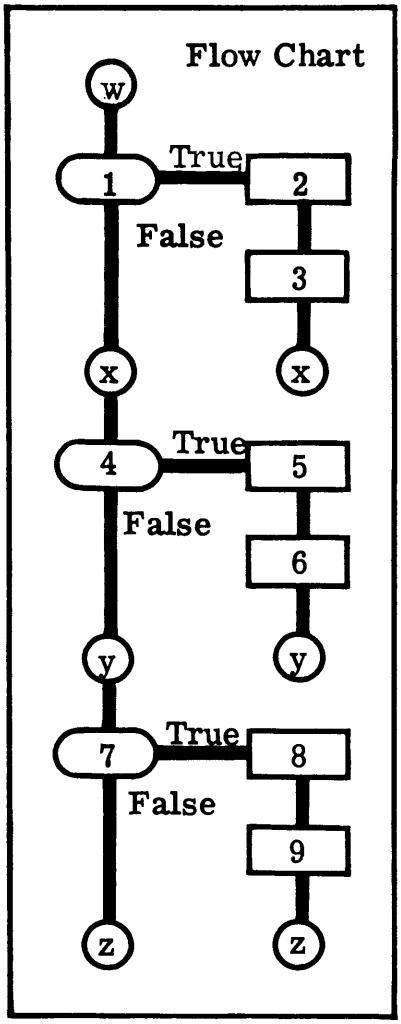
Write JOVIAL statements to:

(a) Compute ALPHA, a floating-point item, equal to the least value of SCORE.

```
BEGIN  COMPUTE'ALPHA.
ALPHA = SCORE($Ø$) $
IF ALPHA GR SCORE($1$) $
     ALPHA = SCORE($1$) $
IF ALPHA GR SCORE($2$) $
     ALPHA = SCORE($2$) $
IF ALPHA GR SCORE($3$) $
     ALPHA = SCORE($3$) $
END
```

(b) Compute BETA, a floating-point item, equal to the sum of all greater than average values of SCORE.

```
BEGIN  COMPUTE'BETA.
ITEM AVERAGE Floating Rounded $
AVERAGE = (SCORE($Ø$)+SCORE($1$)
+ SCORE($2$)+SCORE($3$))/4. $
BETA = Ø. $
IF SCORE($Ø$) GR AVERAGE $
     BETA = BETA+SCORE($Ø$) $
IF SCORE($1$) GR AVERAGE $
     BETA = BETA+SCORE($1$) $
IF SCORE($2$) GR AVERAGE $
     BETA = BETA+SCORE($2$) $
IF SCORE($3$) GR AVERAGE $
     BETA = BETA+SCORE($3$) $
END
```

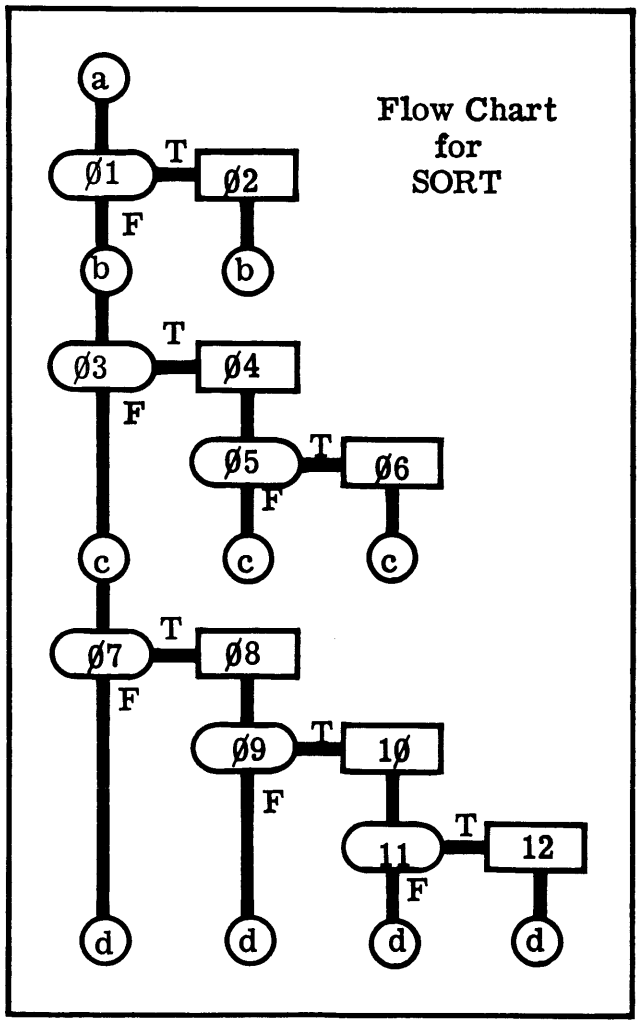


(c) Compute GAMMA, a Boolean item, which is True when the number of greater than average values of SCORE equals the number of lower than average values, and False otherwise.

```
BEGIN  COMPUTE'GAMMA
ITEM HIGH'COUNT fixed 3 Signed $
HIGH'COUNT = Ø $
IF SCORE($Ø$) GR AVERAGE $
     HIGH'COUNT = HIGH'COUNT+1 $
IF SCORE($Ø$) LS AVERAGE $
     HIGH'COUNT = HIGH'COUNT-1 $
IF SCORE($1$) GR AVERAGE $
     HIGH'COUNT = HIGH'COUNT+1 $
IF SCORE($1$) LS AVERAGE $
     HIGH'COUNT = HIGH'COUNT-1 $
IF SCORE($2$) GR AVERAGE $
     HIGH'COUNT = HIGH'COUNT+1 $
IF SCORE($2$) LS AVERAGE $
     HIGH'COUNT = HIGH'COUNT-1 $
IF SCORE($3$) GR AVERAGE $
     HIGH'COUNT = HIGH'COUNT+1 $
IF SCORE($3$) LS AVERAGE $
     HIGH'COUNT = HIGH'COUNT-1 $
GAMMA = HIGH'COUNT EQ Ø $
END
```

(d) Arrange the values of SCORE in ascending numerical order.

```
SORT. BEGIN
SØ1.  IF SCORE($Ø$) GR
      SCORE($1$) $
SØ2.        SCORE($Ø$) ==
      SCORE($1$) $
SØ3.  IF SCORE($1$) GR
      SCORE($2$) $
            BEGIN
SØ4.        SCORE($1$) ==
      SCORE($2$) $
SØ5.        IF SCORE($Ø$)
      GR SCORE($1$) $
SØ6.              SCORE($Ø$)
      == SCORE($1$) $
            END
SØ7.  IF SCORE($2$) GR
      SCORE($3$) $
            BEGIN
SØ8.        SCORE($2$) ==
      SCORE($3$) $
SØ9.        IF SCORE($1$)
      GR SCORE($2$) $
                  BEGIN
S1Ø.              SCORE($1$)
      == SCORE($2$) $
S11.              IF SCORE
      ($Ø$) GR SCORE($1$) $
S12.                    SCORE
      ($Ø$) == SCORE($1$) $
      END  END  END
```



The statement execution flow for SORT shows an interesting regularity, which suggests how the statement might be extended to sort longer lists of numbers

GOTO STATEMENTS. A GOTO statement, composed of the GOTO sequential operator followed by a statement name and terminated by the $ separator breaks the normal listed sequence of statement executions by causing the computer to execute as the next statement the one bearing the given name.

statement ≠ GOTO name$_{\text{of-statement-to-be-executed-next}}$ $

A GOTO statement thus discontinues the execution of a set of consecutively listed statements and initiates the execution of another such set beginning at an explicitly specified statement. Examples of GOTO statements are:

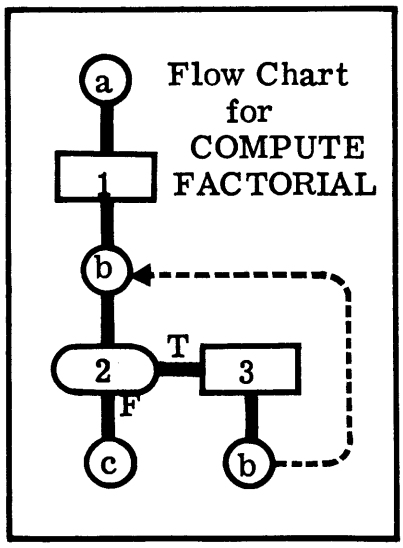
```
GOTO  STEP19 $
GOTO  COMPUTE'TAX $
```

GOTO statements are important in JOVIAL, since they are needed for any departure from the normal, listed sequence of statement executions (except the conditional skipping provided by IF statements). Such departures are required whenever it is impossible, inconvenient, or inefficient to list a statement's successor immediately after it. For example, when a statement must be the unconditional successor to many statements, it can, obviously, be listed after only one of these; all the others must be followed by GOTO statements. As a further example, when different parts of a program are written as separate blocks and must finally be joined into a single program, their proper interleaving in the listing sequence may be difficult to achieve. In such cases, GOTO statements can easily supply the correct execution sequence.

The most noteworthy use of the GOTO statement, however, is to return to an earlier point in the computation and thus form a loop in the sequence of statement executions. Consider the following statement, which computes the factorial of a number.

```
       BEGIN  COMPUTE'FACTORIAL.
STEP1. FACTORIAL = 1 $
STEP2. IF NUMBER GR 1 $
STEP3.      BEGIN  ''Accumulate the product.''
            FACTORIAL = FACTORIAL * NUMBER $
            ''Compute the next factor.''
            NUMBER = NUMBER - 1 $
            GOTO STEP2 $
       END  END
```

The statements are executed in the sequence: STEP1, STEP2, STEP3, STEP2, STEP3, STEP2, STEP3, and so on until the number is reduced to 1 and a final comparison at STEP2 terminates the computation. The loop in the flow of statement executions is shown in the accompanying diagram.



A similar loop may be used in computing arcsin of x from the infinite series summation (where $-1 < x < +1$).

$$\sin^{-1} x = \frac{x}{1} + \frac{1}{2} * \frac{x^3}{3} + \frac{1}{2} * \frac{3}{4} * \frac{x^5}{7} + \frac{1}{2} * \frac{3}{4} * \frac{5}{6} * \frac{x^7}{7} + \ldots$$

In this case, the computation to be repeated consists of (a) accumulating the sum, and (b) computing the next term. Here, the criterion for completing the computation cannot be the exhaustion of the terms, but must rather be: whether or not a particular term appreciably affects the total. If the decision is to disregard all terms less than 1.E-3 in magnitude, then the computation is done when the first term is discarded, since the terms of the series form a sequence of decreasing magnitude. Notice that each term in this series may be computed from the previous term. This computation may be considerably simplified, however, by computing terms in two stages, as a partial term and as a full term, using the previous partial term to compute each new term. The fixed-point variable items ARCSIN, XX, and the necessary temporary items PARTIAL, FULL, and POWER, are declared below, along with the statement computing ARCSIN from XX.

```
ITEM  ARCSIN fixed 16 Signed 1ø Rounded $ ''Angle measured in radiians''
ITEM      XX fixed 16 Signed 15 Rounded $ ''Sine''
ITEM PARTIAL fixed 16 Signed 15 Rounded $ ''Partial term in series''
ITEM    FULL fixed 16 Signed 15 Rounded $ ''Full term in series''
ITEM   POWER fixed 16 Unsigned          $ ''Power of XX in term''
                      BEGIN
                      ARCSIN = ø $
                      PARTIAL = XX $
                      POWER = 1 $
   COMPUTE'FULL'TERM. FULL = PARTIAL/POWER $
                      IF (/FULL/) GR 1.E-3A15 $
                           BEGIN
                           ARCSIN = ARCSIN+FULL $
                           PARTIAL = PARTIAL*XX**2*POWER/(POWER+1) $
                           POWER = POWER+2 $
                           GOTO COMPUTE'FULL'TERM $
                           END
                      END
```

EXERCISE (Basic Statements and Declarations)

(a) The following statement computes a value for EPSILON. Construct a simple assignment statement performing the same function.

```
BEGIN
T1 = 1. $
EPSILON = Ø. $
T1 = T1*OMEGA $
EPSILON = EPSILON+T1+ALPHA($3$) $
T1 = T1*OMEGA $
EPSILON = EPSILON+T1+ALPHA($2$) $
T1 = T1*OMEGA $
EPSILON = EPSILON+T1+ALPHA($1$) $
T1 = T1*OMEGA $
EPSILON = EPSILON+T1+ALPHA($Ø$) $
END
```

(b) Write a JOVIAL statement to compute ARCTAN of X from the infinite series summations:

If $|x| < 1$,

$$\tan^{-1}x = \frac{x}{1} - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots$$

If $|x| = 1$, $\tan^{-1}x = \pi \frac{x}{4}$

If $|x| > 1$,

$$\tan^{-1}x = \frac{1}{2} - \frac{x^{-1}}{1} + \frac{x^{-3}}{3} - \frac{x^{-5}}{5} + \ldots$$

where terms less than 1.E-3 in magnitude may be discarded. Include any necessary declarations, other than

```
ITEM ARCTAN fixed 16 Signed 1Ø $
ITEM     XX fixed 3Ø Signed 15 $
```

(c) Describe the function of the JOVIAL statement below and rewrite it eliminating the items TØ, T1, and T2.

```
BEGIN
MODE Floating Rounded $
TØ = 2.*AA $
R1 = -BB/TØ $
R2 = R1 $
T1 = BB**2-4.*AA*CC $
T2 = (/T1/)**.5/TØ $
IF T1 LS Ø. $
     BEGIN
     I1 = T2 $
     I2 = T2 $
     END
IF T1 GR Ø. $
     BEGIN
     R1 = R1+T2 $
     R2 = R2-T2 $
     END
IF T1 GQ Ø. $
     BEGIN
     I1 = Ø. $
     I2 = Ø. $
     END
END
```

(d) Simplify the following JOVIAL statement.

```
    BEGIN
SØ. OMEGA = Ø $
S1. IF A LS Ø $
S2.      GOTO S6 $
S3. IF B LS Ø $
S4.      GOTO S8 $
S5. GOTO EXIT $
S6. OMEGA = OMEGA+1 $
S7. GOTO S3 $
S8. OMEGA = OMEGA+2 $
S9. GOTO EXIT $
    END
```

(e) Given: the item ALPHA'LENGTH as designating the number of elements in the linear array, ALPHA. The following three statements perform the same function. Describe that function and discuss the differences in its execution.

```
S1. BEGIN
    IF LØ LS Ø $
         BEGIN
         IX = ALPHA'LENGTH+LØ $
         GOTO GET'ANSWER $
         END
    IX = LØ $
GET'ANSWER.
    ANSWER = ALPHA($IX$) $
    END

S2. BEGIN
    IX = LØ $
    IF LØ LS Ø $
         IX = ALPHA'LENGTH+IX $
    ANSWER = ALPHA($IX$) $
    END

S3. BEGIN
    ANSWER = ALPHA($LØ$) $
    IF LØ LS Ø $  ANSWER = ALPHA
    ($ALPHA'LENGTH+LØ$) $
    END
```

(f) Given the JOVIAL statement below, construct reasonable declarations for the items, on the assumption that object K is part of an appliance dealer's inventory.

```
BEGIN
NET'PROFIT($K$) = (PRICE($K$)
- COST($K$))*VOLUME($K$)-OVERHEAD
($K$) $
EFFICIENCY($K$) = NET'PROFIT($K$)
/ INVESTMENT($K$) $
END
```

(g) Given three Boolean items, IND'A, IND'B, and IND'C. Write a JOVIAL statement which selects its successor according to the following chart.

IND'A	IND'B	IND'C	
False	False	False	STEPØ
False	False	True	STEP1
False	True	False	STEP2
False	True	True	STEP3
True	False	False	STEP4
True	False	True	STEP5
True	True	False	STEP6
True	True	True	STEP7

(h) The items INDICATORA, INDICATORB, and INDICATORC are Boolean. Simplify the following JOVIAL statement.

```
        BEGIN
        INDICATORA = Ø $
        INDICATORB = Ø $
        INDICATORC = Ø $
        IF DIFFERENCE LS .ØØ5 $
             GOTO SETABC $
        IF DIFFERENCE LS .Ø5 $
             GOTO SETBC $
        IF DIFFERENCE LS .5 $
             GOTO SETC $
        GOTO NEXT $
SETABC. INDICATORA = 1 $
 SETBC. INDICATORB = 1 $
  SETC. INDICATORC = 1 $
        END
  NEXT.
```

LOOPS

The loop concept, touched on briefly in the previous section, is undoubtedly the single most important concept in computer programming. Loops are important because most modern computers can execute instructions much faster than they can be input, and because the computer is economical only when the same computation is performed on many sets of data.

A loop may be defined as the repeated execution (or, more impressively, as the iterated execution) of a set of statements, usually with some type of modification between repetitions. It would be pointless, of course, to repeat exactly the same computation over and over again; some of the information must be modified between repetitions. Values thus modified become parameters for the loop; that is, they (generally) remain constant during any particular repetition, but vary between repetitions. Loop parameters may be involved in the computation performed by the loop (as were the factors in the factorial loop and the terms in the arcsin loop) or they may be, simply, counters counting the number of repetitions. For the correct functioning of a loop, the loop parameters must initially be assigned the values needed for the first pass through the loop.

To avoid an endless loop, some sort of test must be made to determine when to terminate the repetition. Such tests usually involve one of the loop parameters, so that loops are either count-controlled or condition-controlled, depending on whether or not the termination criterion is the number of repetitions. The previous factorial loop is an instance of a count-controlled loop (where the number of repetitions for N! equals N-1), while the arcsin loop is an instance of a condition-controlled loop (where the condition is: whether the new term appreciably affects the sum).

As seen above, the five basic functions of a loop are to:

1. Initialize the loop parameters;

2. Execute the set of statements;

3. Modify the loop parameters; and

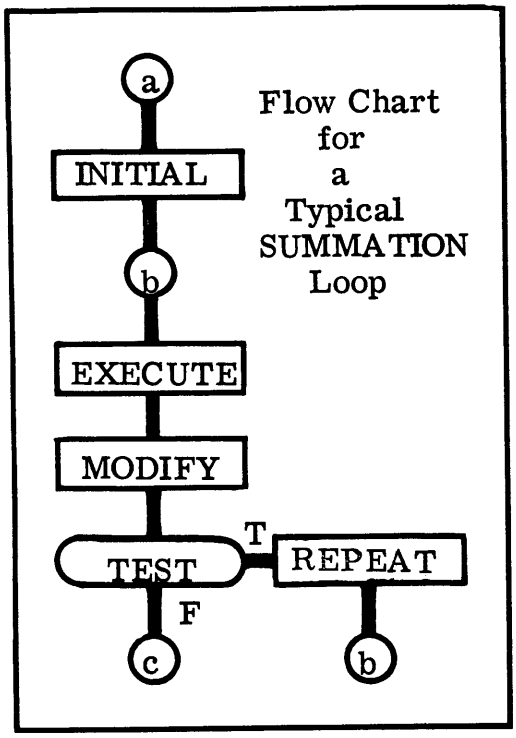
4. Test the controlling loop parameter, to determine whether to

5. Repeat the execution.

It should be realized, however, that the order of these five steps is not necessarily always the same as listed above, for it is often desirable to change the order of Execute, Modify and Test. The factorial loop, for example, exhibits the pattern: Initialize; Test; Execute; Modify; Repeat, while the arcsin loop exhibits the pattern: Initialize; Modify; Test; Execute; Modify; Repeat.

The five basic functions of a loop are very clearly illustrated in the almost classic loop example, computing the sum of 50 numbers. A JOVIAL version of this computation appears below, along with the necessary data declarations.

```
ARRAY ALPHA 50 Floating Rounded $
ITEM    SUM Floating Rounded $
ITEM  COUNT fixed 6 Unsigned $
            BEGIN SUMMATION.
INITIALIZE. COUNT = 0 $ SUM = 0. $
   EXECUTE. SUM = SUM+ALPHA($COUNT$) $
    MODIFY. COUNT = COUNT+1 $
      TEST. IF COUNT LQ 49 $
    REPEAT.      GOTO EXECUTE $
            END
```



The above loop is, of course, a count-controlled loop, and the device of running the count from 0 thru 49 allows it to both control the loop and index the array item, ALPHA. The statement execution flow is shown in the accompanying diagram.

FOR STATEMENTS. Counting and count-controlled loops are among the most common operations in programming. For this reason, JOVIAL includes a special set of signed, integer valued variables -- subscripts, designated by single letters. Subscripts are activated and assigned initial values by the execution of FOR statements. They are eminently suited for use as counters, since an active subscript can be used for exactly the same purposes as any other numeric variable.

Only an inactive subscript may be activated by a FOR statement and, in general, the range of a subscript's activity is bounded by the activating FOR statement and by the next (non-FOR) statement listed. Outside this range, a subscript is inactive, and references to it are undefined. Furthermore, since it is the execution of the FOR statement that activates the subscript, a GOTO statement entering the subscript's nominal range of activity from outside will find it inactive and its value undefined.

ONE-FACTOR FOR STATEMENTS. The simplest FOR statement, composed of the FOR sequential operator followed by a subscript letter, the = separator, and a numeric formula, and terminated by the $ separator, merely activates the subscript and assigns it the value specified by the formula.

statement ↓ FOR letter = numeric-formula$_{\text{initial-value}}$ $

The subscript is active and may be used as a numeric variable until the end of the next (non-FOR) statement listed, which may, of course, be compound. For example:

```
ARRAY ALPHA 5Ø Floating Rounded $
ARRAY  BETA 5Ø Floating Rounded $
ARRAY GAMMA 5Ø Boolean $
          BEGIN
INITIALIZE. FOR C = Ø $
                BEGIN
   EXECUTE.     GAMMA($C$) = ALPHA($C$) EQ BETA($C$) $
    MODIFY.     C = C+1 $
      TEST.     IF C LQ 49 $
    REPEAT.         GOTO EXECUTE $
          END  END
```

This statement assigns the value True to those GAMMAs corresponding by index to equal values of ALPHA and BETA, and False to all other GAMMAs. The structure of this loop is identical to that of the loop for summing the ALPHAs, but the explicit declaration of a count item is unnecessary, since this function is assumed by the subscript, C.

Subscripts can, of course, be used for purposes other than counting. The following example, which assumes the activity of subscripts A and Z, transfers control to statement STEP7 if the smaller of the integers designated by A and Z is a factor of the other.

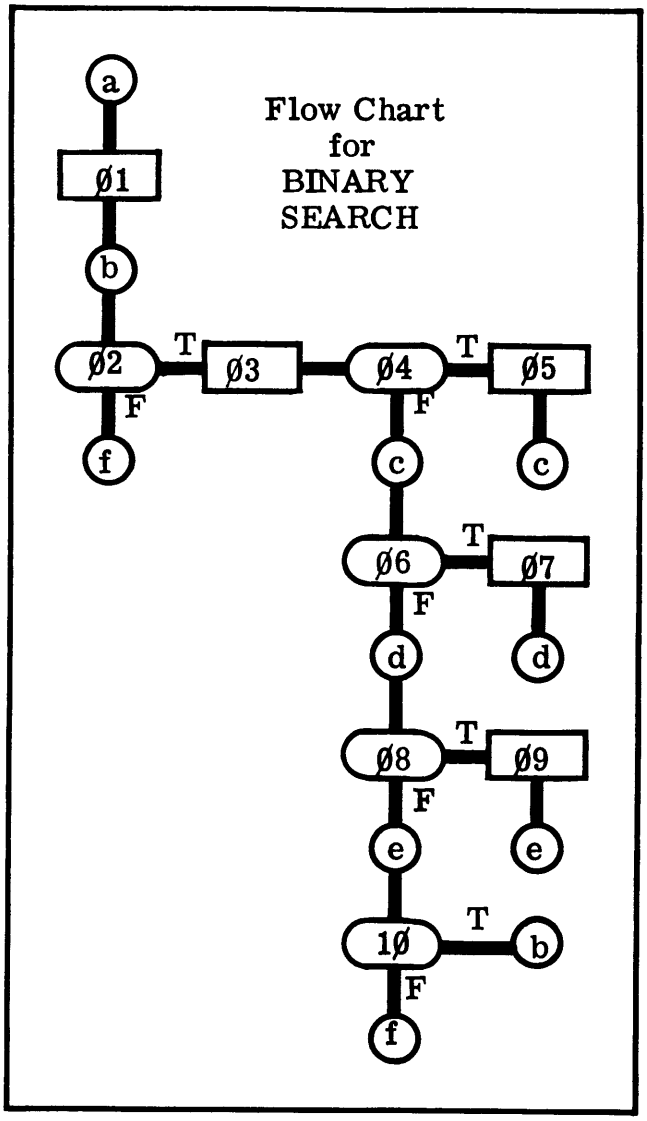
```
BEGIN
IF (/A/) LS (/Z/) $
     A == Z $
IF Z NQ Ø $
     BEGIN
     FOR K = A/Z $
          IF K EQ A/Z $
               GOTO STEP7 $
     END
END
```

Note: The FOR statement activating the subscript K truncates any fractional remainder from the division, whereas the comparison resulting from the relational operator EQ does not.

The following example requires that some initial conditions be assumed. Given: two linear arrays, QUESTION and ANSWER, where the QUESTIONs are arranged in ascending numeric order and correspond by index to the ANSWERs; an item, QUE, of the same type as the QUESTIONs; and two active subscripts, A and Z, which designate the indexes of the first and last question-answer pair to be considered. Search the bounded array of QUESTIONs for one designated by QUE and if successful, set the Boolean item SUCCESS to True and the item ANS to the corresponding ANSWER, or if unsuccessful, set SUCCESS to False.

```
                BEGIN BINARY'SEARCH
        STEPØ1. SUCCESS = Ø $
SEARCH. STEPØ2. IF A LQ Z $
                     BEGIN
        STEPØ3.      FOR K = ''The midpoint,'' (A+Z)/2 $
                          BEGIN
        STEPØ4.           IF QUE EQ QUESTION($K$) $
        STEPØ5.                BEGIN
                               ANS = ANSWER($K$) $
                               SUCCESS = 1 $
                               END
                          ''If QUE NQ QUESTION($A...K$), then
                          search QUESTION($K+1...Z$).  That is''
        STEPØ6.           IF QUE GR QUESTION($K$) $
        STEPØ7.                A = K+1 $
                          ''If QUE NQ QUESTION($K...Z$), then
                          search QUESTION($A...K-1$).  That is''
        STEPØ8.           IF QUE LS QUESTION($K$) $
        STEPØ9.                Z = K-1 $
        STEP1Ø.           IF NOT SUCCESS $
                               GOTO SEARCH $
                END  END  END
```

The search technique described above is usually referred to as a "binary search," because each pass through the loop splits the searched list in two and, due to its ascending order, discards that half of it which does not include the value sought. The execution of this statement is shown graphically in the accompanying flow chart.



TWO-FACTOR FOR STATEMENTS. The two-factor FOR statement, composed of the FOR sequential operator or followed by a subscript letter, the = separator, and two numeric formulas separated by the , separator, and terminated by the $ separator, activates the subscript, assigns it the value specified by the first numeric formula, and causes the (non-FOR) statement following it to be repeatedly executed an indefinite number of times. After each repetition, the value of the subscript is incremented by the value specified by the second numeric formula. A two-factor FOR statement is useful, therefore, in condition-controlled loops where one of the non-controlling loop parameters is a count.

statement ♯ FOR letter = numeric-formula$_{\text{initial-value}}$, numeric-formula$_{\text{increment}}$ $

The operation of the two-factor FOR statement is best explained in terms of simpler statements. In effect, then,

```
FOR L = INITIAL, INCREMENT $
     BEGIN
     .
     .
     .
     END
```

is just a shorthand way of writing

```
INITIALIZE. FOR L = INITIAL $
   EXECUTE.      BEGIN
                 .
                 .
                 .
                      BEGIN
    MODIFY.           L = L + INCREMENT $
    REPEAT.           GOTO EXECUTE $
                      END
                 END
```

The addition of a second numeric formula, forming a two-factor FOR statement, automatically implies the compound, MODIFY-and-REPEAT statement, so that the set of repeating statements should be written with the existence of this implicit statement in mind. In particular, since the two-factor FOR statement, by itself, creates an endless loop, a terminating test must be supplied among these statements. For example:

```
ARRAY ALPHA 5Ø Floating Rounded $
ARRAY  BETA 5Ø Floating Rounded $
ARRAY GAMMA 5Ø Boolean $
            BEGIN
            FOR C = Ø,1 $
                 BEGIN
                 IF NOT GAMMA($C$) $
                      BEGIN
                      ALPHA($C$) = Ø. $
                      BETA($C$)  = Ø. $
                      END
                 IF C LS 49 $
                      ''Implicit MODIFY-and-REPEAT statement
                      inserted at this point.''
                 END
            END
```

The above statement assigns zero as the value of those ALPHAs and BETAs corresponding by index to False values of GAMMA. The IF statement ultimately terminates the loop by skipping the implicitly inserted MODIFY-and-REPEAT statement after C has reached a value of 49.

To further illustrate the two-factor FOR statement, the evaluation of the hyperbolic sine from the series summation,

$$\sinh x = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots,$$

provides a good example of a condition-controlled loop with a counter as a non-controlling loop parameter. Terms less than .ØØ5 in magnitude are disregarded.

```
ITEM SINH fixed 16 Signed Ø8 Rounded Ø...75 $
ITEM   XX fixed 16 Signed 12 Rounded Ø...Ø5 $
ITEM TERM fixed 16 Signed 1Ø Rounded Ø...3Ø $
ITEM   X2 fixed 16 Signed 1Ø Rounded Ø...25 $
          BEGIN
          X2 = XX ** 2 $
          SINH = Ø $
          TERM = XX $
          FOR P = 3,2 $
               BEGIN
               SINH = SINH + TERM $
               TERM = TERM * X2 / (P * (P - 1)) $
               IF (/TERM/) GQ .5E-2A8 $
               END
          END
```

The temporary storage item, X2, allows x^2 to be computed once and not each pass through the loop, since it is not a loop parameter.

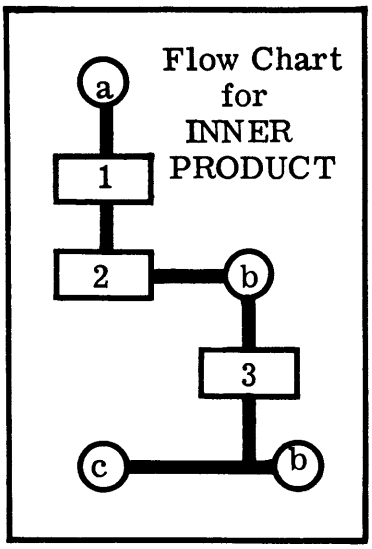
COMPLETE FOR STATEMENTS. The complete, three-factor FOR statement, composed of the FOR sequential operator followed by a subscript letter, the = separator, three numeric formulas separated by the , separator, and terminated by the $ separator, activates the subscript, assigns it the value specified by the first numeric formula, and causes the (non-FOR) statement following it to be repeatedly executed a definite number of times. After each repetition, the value of the subscript is incremented by the value specified by the second numeric formula, and this modified value is compared with the limiting value specified by the third numeric formula to determine whether to terminate the loop. A complete FOR statement thus produces a count-controlled loop.

statement ∮ FOR letter = numeric-formula$_{\text{initial-value}}$, numeric-formula$_{\text{increment}}$, numeric-formula$_{\text{limit-value}}$ $

Before discussing the iteration mechanism created by the complete FOR statement, it would be well to consider a simple example: the evaluation of the inner or dot product of the two numeric vectors, ALPHA and BETA.

```
ARRAY ALPHA 5Ø Floating Rounded $
ARRAY  BETA 5Ø Floating Rounded $
ITEM    DOT    Floating Rounded $
          BEGIN INNER'PRODUCT.
     STEP1. DOT = Ø. $
     STEP2. FOR I = Ø,1,49 $
     STEP3.       DOT = DOT+ALPHA($I$)*BETA($I$) $
          END''INNER'PRODUCT''
```

STEP3, the repeating statement of the loop, is executed 5Ø times, as the subscript I is stepped from an initial value of Ø in increments of 1 to the limit of 49. The accompanying flow chart shows the statement execution sequence. (The branch in the flow line is due to the implicit subscript test created at that point by the complete FOR statement, STEP2.) Notice that, since the increment factor in a FOR statement can be negative as well as positive,




```
STEP2. FOR I = 49, -1, Ø $
```

could also be used in the dot product computation above.

The complete FOR statement creates a complete, count-controlled loop, with INITIALIZE, EXECUTIVE, MODIFY, TEST, and REPEAT steps. The iteration mechanism consists of an implicit and compound MODIFY-TEST-REPEAT statement which is automatically inserted by the compiler after the repeating statements. This means that these statements will always be executed at least once. Whether they are executed more than once depends, of course, on the outcome of the implied subscript test following the subscript modification. This test compares the subscript's modified value with the limiting value specified by the third numeric formula in the FOR statement. The nature of this comparison depends on the second, incremental, numeric formula of the FOR statement, and whether it specifies a negative value. If it specifies a decrement or negative increment, the loop is repeated only if the subscript's value

is greater than or equal to the limit value. If, on the other hand, it specifies a positive (non-negative) increment, the loop is repeated only if the subscript's value is less than or equal to the limit value. Thus, the loop is terminated when the subscript's modified value passes the limit value, either in the positive direction for a positive increment, or in the negative direction for a negative increment.

As with the two-factor FOR statement, the operation of the complete FOR statement can be explained in terms of simpler statements. In effect, then,

```
FOR L = INITIAL, INCREMENT, LIMIT $
     BEGIN
     .
     .
     END
```

is just a shorthand way of writing

```
INITIALIZE. FOR L = INITIAL $
   EXECUTE.       BEGIN
                  .
                  .
                       BEGIN
    MODIFY.            L = L + INCREMENT $
      TEST.            IF (INCREMENT GQ Ø AND L LQ LIMIT)
                       OR (INCREMENT LS Ø AND L GQ LIMIT) $
    REPEAT.                 GOTO EXECUTE $
                       END
                  END
```

The subscript test shown above can often be simplified. When the increment formula cannot specify a negative value (as with a positive constant or an unsigned variable) the test is reduced to

```
TEST. IF L LQ LIMIT $  REPEAT. GOTO EXECUTE $
```

and when the increment formula cannot specify a positive value (as with a negative constant) the test is

```
TEST. IF L GQ LIMIT $  REPEAT. GOTO EXECUTE $
```

Consider some further examples of count-controlled loops created by complete FOR statements.

The following statement exchanges the first 25 ALPHA value pairs.

```
BEGIN
FOR I = ∅,2,48 $
     ALPHA($I$) == ALPHA($I+1$) $
END
```

The following statement evaluates a polynomial of a given degree whose coefficients, corresponding by index to the powers of the variable XX, are elements of the linear array, ALPHA.

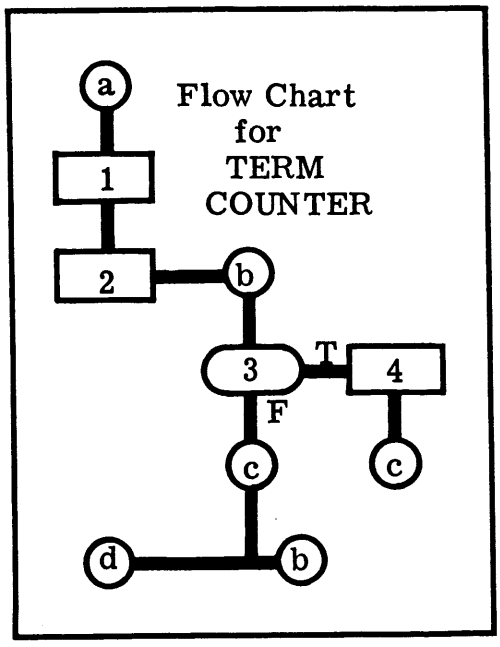
```
POLYVAL. BEGIN
  STEP1. POLYNOMIAL = ∅. $
  STEP2. FOR P = DEGREE, -1, ∅ $
  STEP3.       POLYNOMIAL = POLYNOMIAL**XX+ALPHA($P$) $
         END  ''POLYVAL''
```

In this loop, STEP3, the repeating statement, is executed DEGREE+1 times.

Given a JOVIAL program in the form of a linear array of Hollerith coded, single character, literal values; the following statement counts the number of $ separators (ignoring subscript brackets).

```
  ARRAY   CHARACTER 65536 Hollerith 1 $''JOVIAL program, 2**16 signs max.''
  ITEM    TERM'COUNT fixed 16 Unsigned $''Number of termination separators''
  ITEM N'CHARACTERS fixed 16 Unsigned $
                    BEGIN TERM'COUNTER.
              STEP1. TERM'COUNT = ∅ $
              STEP2. FOR C = ∅,1,N'CHARACTERS-1 $
                           BEGIN
              STEP3.       IF CHARACTER($C$) EQ 1H($) AND ((C NQ ∅ AND
CHARACTER($C-1$) NQ 1H(()) OR (C NQ N'CHARACTERS-1 AND CHARACTER($C+1$)
NQ 1H())) $  STEP4.            TERM'COUNT = TERM'COUNT+1 $
                           END
                    END
```

The accompanying flow chart graphically illustrates the statement execution sequence of this routine.



The following statement evaluates the algebraic formula

$$A_i = \frac{B_i x_i}{\sqrt{\sqrt{(2.3826C_i - 1/C_i) + D^i}}}$$

for i = ∅ thru 999, where x_i = .∅∅1(i+1).

```
ARRAY AA 1ØØØ Floating Rounded $''Values of A''
ARRAY BB 1ØØØ Floating Rounded $''Values of B''
ARRAY CC 1ØØØ Floating Rounded $''Values of C''
ITEM  DD      Floating Rounded $''Value  of D''
ITEM  PD      Floating Rounded $''Powers of D''
ITEM  XX      Floating Rounded $''Values of x''
              BEGIN
              XX = .ØØ1 $ PD = 1. $
              FOR I = Ø,1,999 $
                   BEGIN
                   AA($I$) = BB($I$)*XX/((2.3826*CC($I$)
-1./CC($I$))+PD)**.5 $
                   PD = PD*DD $
                   XX = XX+.ØØ1 $
                   END
              END
```

As still a further example, consider the following statement.

```
ARRAY CHARACTER 2ØØØØ Hollerith 1 $
ITEM     LENGTH fixed 15 Unsigned Ø...2ØØØØ $
        REDUCE. ''A statement that shortens a given length string of
literal characters by reducing strings of blanks to single blanks.''
BEGIN
IF LENGTH GR Ø $
     BEGIN
     FOR L = LENGTH-1 $
          BEGIN
ITEM BLANKS Boolean $''True means that the last character was a blank''
          BLANKS = 1 $''Which eliminates initial blanks''
          LENGTH = Ø $
          FOR I = Ø,1,L $
SKIP.          BEGIN
               IF NOT BLANKS OR CHARACTER($I$) NQ 1H( ) $
                    BEGIN
                    BLANKS = NOT BLANKS AND CHARACTER($I$) EQ 1H( ) $
                    CHARACTER($LENGTH$) = CHARACTER($I$) $
                    LENGTH = LENGTH+1 $
END  END  END  END  END
```

The action taken by SKIP, the repeated statement in the above loop, is tabulated below.

CHARACTER(I)	BLANKS	BLANKS	CHARACTER($LENGTH$)	LENGTH
EQ 1H( )	TRUE			
NQ 1H( )	TRUE	= FALSE	= CHARACTER(I)	= LENGTH+1
EQ 1H( )	FALSE	= TRUE	= CHARACTER(I)	= LENGTH+1
NQ 1H( )	FALSE		= CHARACTER(I)	= LENGTH+1

It should be realized that the factors of a FOR statement are not always constants as in most of the previous examples, but are occasionally variables or even numeric formulas. The loops created by such FOR statements, though simple in structure, can be quite complex in operation, especially if the repeating statements themselves modify the subscript's value or the values specified by either the increment factor or the limit factor.

For example, consider the following problem and its JOVIAL solution: The 1ØØØ-element, linear array TERM consists of several interleaved but separate lists of numbers. Another linear array, STEP, corresponds by index to the array TERM, so that whenever TERM(N) is a member of a list, then STEP(N) is the index increment and N+STEP(N) is the index of the next member of the list; except when TERM(N) is the last member of a list, and then N+STEP(N) specifies 1ØØØ. The linear arrays FIRST, SUM, and AVERAGE correspond by index and provide information about the various lists. Assuming subscript L is active and FIRST(L) designates the TERM-index of the first member of list L, then it is necessary to compute SUM(L), the sum of all the members of the list, and AVERAGE(L), their average.

```
ARRAY   TERM 1ØØØ Floating Rounded $
ARRAY   STEP 1ØØØ fixed 1Ø Unsigned $
ITEM MEMBERS      Floating Rounded $''Number of members in a list''
             BEGIN
             SUM($L$) = Ø. $  MEMBERS = Ø. $
             FOR N = FIRST($L$),STEP($N$),999 $
                  BEGIN
                  SUM($L$) = SUM($L$)+TERM($N$) $
                  MEMBERS = MEMBERS+1. $
                  END
             AVERAGE($L$) = SUM($L$)/MEMBERS $
             END
```

As another example, consider the solution to the following problem: A certain language consists entirely of 4-letter words, which are separated from each other by one or more blanks. In this language, a well-formed sentence of a given length is represented by a linear array of literal characters. It is necessary to determine the number of words in such a sentence.

```
ARRAY CHARACTER 1ØØØØ Hollerith 1 $''The sentence.''
ITEM     LENGTH fixed 14 Unsigned 1...1ØØØØ $''Number of characters''
ITEM      WORDS fixed 14 Unsigned 1...2ØØØ  $''Number of words''
               BEGIN
               WORDS = Ø $
               FOR K = Ø,1,LENGTH-4 $
                    BEGIN
                    IF CHARACTER($K$) NQ 1H( ) $
                         BEGIN
                         WORDS = WORDS+1 $
                         K = K+4 $
                         END
                    END
               END
```

LOOPS WITHIN LOOPS. It is often necessary to construct a program loop that is itself repeated a number of times, in other words, a loop within a loop. An example is an operation on each of the elements of a matrix, which is repeated for each element in a row, and then for each row in the matrix. Loops within loops may be constructed by placing the inner loop within the compound statement being repeated by the outer loop.

To illustrate this concept, consider the following statement, which searches the square array, NODE, for its largest absolute value.

```
ARRAY NODE 1ØØ 1ØØ fixed 64 Signed 32 $
ITEM  AMAX         fixed 64 Signed 32 $
          BEGIN
          AMAX = Ø $
          FOR I = Ø,1,99 $
               BEGIN
               FOR J = Ø,1,99 $
                    BEGIN
                    IF (/NODE($I,J$)/) GR AMAX $
                         AMAX = (/NODE($I,J$)/) $
          END  END  END
```

As another example, the statement below divides off-diagonal elements of each row of the same matrix by the diagonal element of that row, if non-zero.

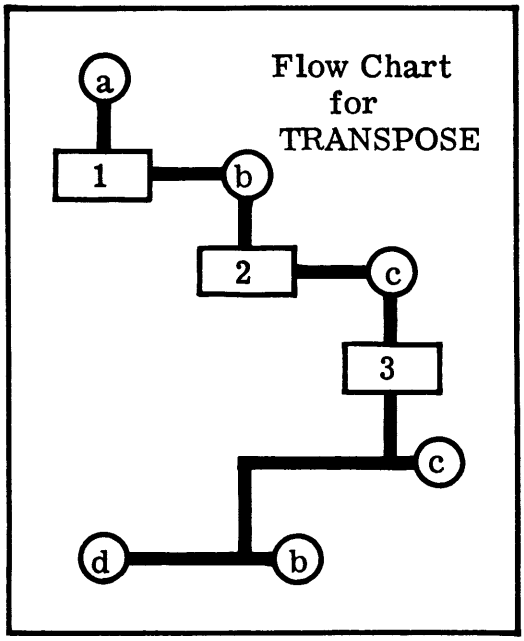
```
BEGIN
FOR I = Ø,1,99 $
     BEGIN
     FOR J = Ø,1,99 $
          BEGIN
          IF I NQ J AND NODE($I,J$) NQ Ø.A32 NQ NODE($I,I$) NQ 1.A32 $
               NODE($I,J$) = NODE($I,J$)/NODE($I,I$) $
END  END  END
```

In both of the above examples, it can be seen that the IF statement repeated by the inner loop is executed 1ØØ times for each repetition of the outer loop, which is itself executed 1ØØ times, thus making a total of 1ØØØØ repetitions.

As a further example, consider the following matrix transposition routine, which interchanges the rows and columns of the same array, NODE.

```
TRANSPOSE. BEGIN
    STEP1. FOR I = Ø,1,99 $
                BEGIN
    STEP2.      FOR J = I+1,1,99 $
    STEP3.           NODE($I,J$) == NODE($J,I$) $
                END
           END
```

Notice in STEP2: The initial factor of the inner FOR statement loop varies with each repetition of the outer loop, so that only unexchanged non-diagonal element pairs are exchanged. The statement execution sequence for this routine, which is similar in loop structure to the other two, is shown in the accompanying flow chart.



The following JOVIAL statement computes new values for each of the elements of the array, NODE. The new value for an element consists of the average value of the horizontally and vertically adjacent elements. Note that a corner element has only two

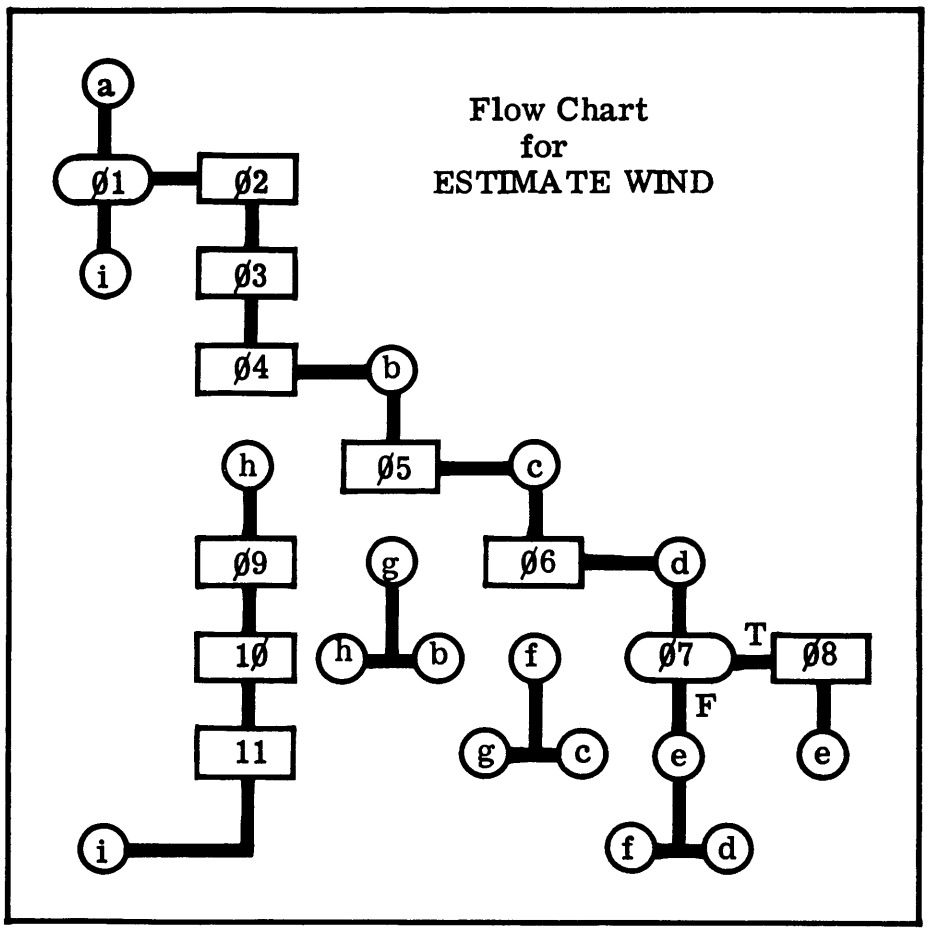
adjacent elements, while other exterior elements have three, and interior elements have four. An array of temporary storage must be declared since no element of NODE can be reset until it is used in computing new values for all elements adjacent to it.

```
ARRAY NODE 1ØØ 1ØØ fixed 64 Signed 32 $
ARRAY TEMP 1ØØ 1ØØ fixed 64 Signed 32 $
ITEM  N'AE         fixed Ø3 Unsigned  $''Number of Adjacent Elements''
          BEGIN
          FOR I = Ø,1,99 $
               BEGIN
               FOR J = Ø,1,99 $
                    BEGIN
                    N'AE = Ø $
                    TEMP($I,J$) = Ø.A32 $
                    IF I NQ Ø $
                         BEGIN
                         N'AE = N'AE+1 $
                         TEMP($I,J$) = TEMP($I,J$)+NODE($I-1,J$) $
                         END
                    IF I NQ 99 $
                         BEGIN
                         N'AE = N'AE+1 $
                         TEMP($I,J$) = TEMP($I,J$)+NODE($I+1,J$) $
                         END
                    IF J NQ Ø $
                         BEGIN
                         N'AE = N'AE+1 $
                         TEMP($I,J$) = TEMP($L,J$)+NODE($I,J-1$) $
                         END
                    IF J NQ 99 $
                         BEGIN
                         N'AE = N'AE+1 $
                         TEMP($I,J$) = TEMP($I,J$)+NODE($I,J+1$) $
                         END
                    TEMP($I,J$) = TEMP($I,J$)/N'AE $
                    END
               END
          FOR I = Ø,1,99 $
               BEGIN
               FOR J = Ø,1,99 $
                    NODE($I,J$) = TEMP($I,J$) $
               END
          END
```

The following statement also computes new values for the elements of an array, based on the values of adjacent elements.

```
     ESTIMATE'WIND. BEGIN''A routine to update the estimate of an un-
                    reported current wind vector of given coordinates,
                    in a wind grid indexed by latitude, longitude, and
                    altitude, by adding the weighted average difference
                    between available adjacent current wind vectors and
                    corresponding seasonal wind vectors to the seasonal
                    wind vector with the given coordinates.''
ARRAY         REPORT 3ø 6ø 6 Status V(UNAVAILABLE) V(ESTIMATED)
                             V(AVAILABLE) $
ARRAY           TIME 3ø 6ø 6 fixed 6 Signed ø...23''hours past midnight
                             zulu time.  Plus=later, Minus=earlier''$
ARRAY SEASONAL'WIND 3ø 6ø 6 Dual 12 Signed 3''in knots East,North''$
ARRAY  CURRENT'WIND 3ø 6ø 6 Dual 12 Signed 3''in knots East,North''$
ITEM INCREMENT'WIND         Dual 12 Signed 3''in knots East,North''$
ITEM            LAT         fixed 6 Unsigned ø...29''degrees North of 22
                            degrees North latitude''$
ITEM            LON         fixed 6 Unsigned ø...59''degrees East of 127
                            degrees West longitude''$
ITEM            ALT         fixed 6 Unsigned ø...ø5''1øøøø ft. intervals
                            above 1øøøø ft. altitude''$
ITEM          CLOCK         fixed 6 Unsigned ø...23''hours past midnight
                            zulu time''$
ITEM       REPORTED         Boolean $
            STEPø1. IF REPORT($LAT,LON,ALT$) NQ V(AVAILABLE) $
                          BEGIN
            STEPø2.       INCREMENT'WIND = D(ø,ø) $
            STEPø3.       REPORTED = REPORT($LAT,LON,ALT$) EQ
V(AVAILABLE) $
            STEPø4.       FOR I = LAT-1,1,LAT+1 $
                               BEGIN
            STEPø5.            FOR J = LON-1,1,LON+1 $
                                    BEGIN
            STEPø6.                 FOR K = ALT-1,1,ALT+1
                                         BEGIN
            STEPø7.                      IF ø LQ I LS 3ø AND ø LQ J LS 6ø
AND ø LQ K LS 6 AND REPORT($I,J,K$) NQ V(UNAVAILABLE) AND (REPORTED OR
TIME($LAT,LON,ALT$) LQ TIME($I,J,K$)) $      BEGIN
            STEPø8.                           INCREMENT'WIND = INCREMENT'
WIND + (CURRENT'WIND($I,J,K$)-SEASONAL'WIND($I,J,K$)) $
                                    END  END  END  END
            STEPø9.       CURRENT'WIND($LAT,LON,ALT$) = INCREMENT'WIND /
D(27,27) + SEASONAL'WIND($LAT,LON,ALT$) $
            STEP1ø.       REPORT($LAT,LON,ALT$) = V(ESTIMATED) $
            STEP11.       TIME($LAT,LON,ALT$) = CLOCK $
                          END  END
```

The flow chart for ESTIMATE'WIND is given below.



The statement below computes values of y from the algebraic formula

$$y_{ijk} = \frac{x_i}{\sqrt{\sqrt{R^{2j} + (2.3826F_k - 1/F_k)}}}$$

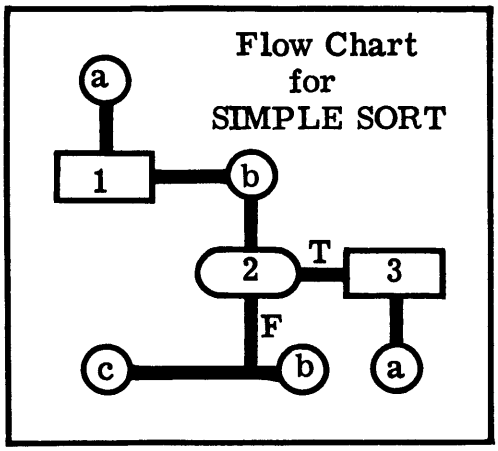
```
ARRAY YY 1ØØ 5Ø 1Ø Floating Rounded $
ARRAY FF        1Ø Floating Rounded $
ITEM  RR           Floating Rounded $
ITEM  XX           Floating Rounded $
         BEGIN
         XX = .ØØ1 $
         FOR I = Ø,1,99 $
              BEGIN
              FOR J = Ø,1,49 $
                   BEGIN
                   FOR K = Ø,1,9 $
                        BEGIN
                        YY($I,J,K$) = XX/(RR**(2*J)+2.3826*FF($K$)
- 1./FF($K$))**.5 $      END
                   END
              XX = XX+.ØØ1 $
              END
         END
```

Multiple loops need not always be embedded one within the other, as in the previous examples. Consider the following statement, which arranges the 1ØØØ, five-character, Transmission-coded literal values of the linear array, WORD, into alphabetic order by searching the array for out-of-order adjacent pairs (STEP2). When such a pair is discovered, it is exchanged into correct order (STEP3) and the search begins over, to terminate only when all adjacent pairs (and thus the entire array) are correctly ordered.

```
ARRAY WORD 1ØØØ Transmission 5 $
           BEGIN SIMPLE'SORT.
    STEP1. FOR I = Ø,1,998 $
                BEGIN
    STEP2.      IF WORD($I$) GR WORD($I+1$) $
    STEP3.           BEGIN
                     WORD($I$) == WORD($I+1$) $
                     GOTO STEP1 $
                     END
                END
           END
```

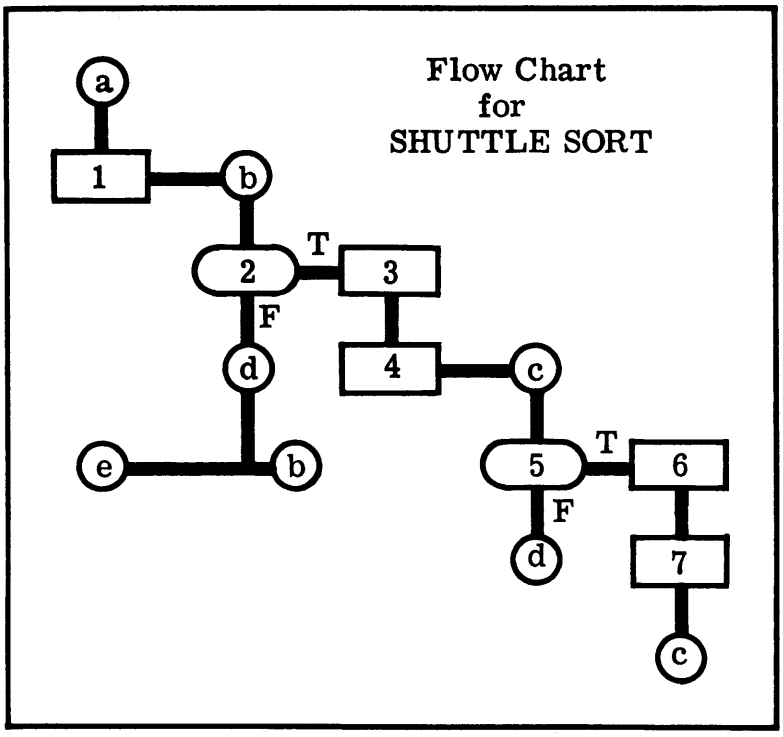
This sorting technique is probably the simplest of all sorts, though it is by no means the most efficient. Notice the existence of two loops, as shown in the accompanying flow chart.

A much more efficient sorting technique, the shuttle exchange, is exemplified in the following statement, which, using the loop within a loop structure, also arranges the literal values of the linear array, WORD, into alphabetic order.




```
      BEGIN SHUTTLE'SORT.
STEP1. FOR I = Ø,1,998 $
            BEGIN
STEP2.      IF WORD($I$) GR WORD($I+1$) $
                  BEGIN
STEP3.            WORD($I$) == WORD($I+1$) $
STEP4.            FOR J = I $
                        BEGIN
STEP5.                  IF J NQ Ø AND WORD($J-1$) GR WORD($J$) $
                              BEGIN
STEP6.                        WORD($J-1$) == WORD($J$) $
STEP7.                        J = J-1 $
                              GOTO STEP5 $
      END  END  END  END  END
```

This statement also searches down the array for out-of-order, adjacent pairs. When such a pair is discovered, its values are exchanged and, in this case, the exchanging continues up the array as far as necessary. The search then resumes where it left off, leaving a correctly ordered set of values behind. The statement execution sequence is shown in the following flow chart.

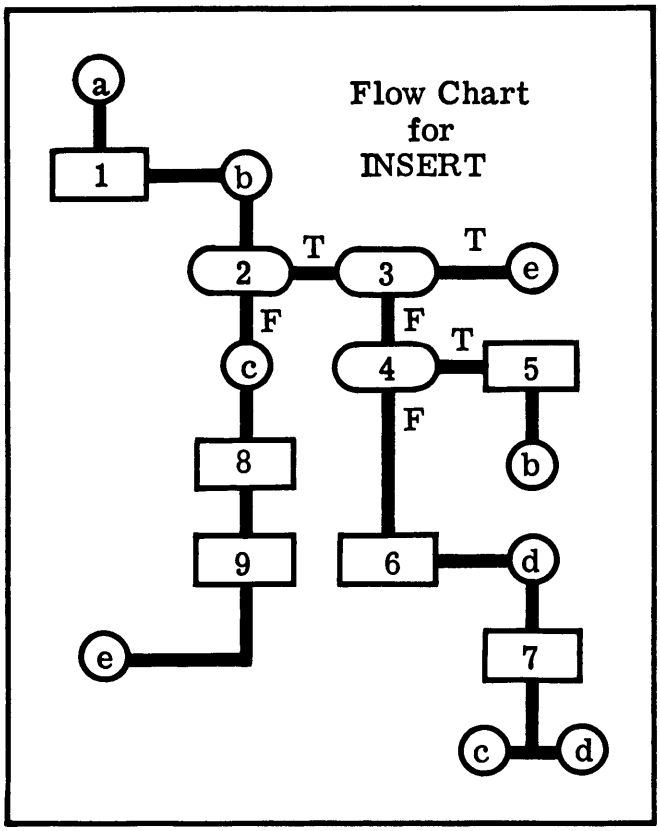


As a further example of a loop within a loop, consider the following statement, INSERT.

```
ARRAY     WORD 1ØØØ Transmission 5 $
ITEM     WORDS      fixed 1Ø Unsigned Ø...1ØØØ $''Number of words''
ITEM NEW'WORD      Transmission 5 $
      INSERT. ''A statement that inserts a new, 5-letter word into
              an alphabetically ordered list of 5-letter words
              whenever the new word is not already listed.''
              BEGIN
       STEP1. FOR K = Ø $
                    BEGIN
       STEP2.       IF K LS words $
                          BEGIN
       STEP3.             IF NEW'WORD EQ WORD($K$) $
                                GOTO CONTINUE $
       STEP4.             IF NEW'WORD GR WORD($K$) $
                                BEGIN
       STEP5.                   K = K+1 $  GOTO STEP2 $
                                END
       STEP6.             FOR J = WORDS,-1,K+1 $
       STEP7.                   WORD($J$) = WORD($J-1$) $
                          END
```

```
        STEP8.      WORD($K$) = NEW'WORD $
        STEP9.      WORDS = WORDS+1 $
                    END
                END  ''INSERT''
    CONTINUE.
```

The execution sequence for this statement is shown in the following flow chart.



FOR-STATEMENT STRINGS. As was previously mentioned, a subscript's range of activity begins with the activating FOR statement and extends over intervening FOR statements to include the first non-FOR statement listed. This allows many subscripts to be conveniently and, in effect, simultaneously activated by a string of FOR statements. The following diagram illustrates the activity ranges of three subscripts, A, B, and C, as supplied by a FOR-statement string.

```
:.........FOR A =  ---  $
:  :......FOR B =  ---  $
:  :  :...FOR C =  ---  $
:  :  :          BEGIN
A  B  C          :
:  :  :          :
:..:..:..........END
```

A FOR-statement string consists of an unbroken sequence of FOR statements, and may contain any number of incomplete, one- and two-factor FOR statements. Only one complete, three-factor FOR statement can appear in a FOR-statement string, however, and it must be listed before any two-factor FOR statements in the string. A FOR-statement string containing increment factors creates a single loop by means of an implicit iteration mechanism which includes a subscript modification for each increment. And if the string begins with a complete FOR statement, a corresponding subscript test is also included. This means that the implicit, compound, subscript MODIFY, TEST, and REPEAT statement inserted at the bottom of the loop will modify the values of all subscripts activated by two or three-factor FOR statements, but will test only the value of that subscript activated by the single, complete, three-factor FOR statement that is listed before the others. Subscript modifications are supplied in reverse of the order in which the subscripts are activated. Consequently, when a subscript test is included, the modification of that subscript immediately precedes it. For example:

```
FOR A = ∅,1,99 $
FOR B = 1,1 $
FOR C = ∅,B $
       BEGIN
       :
       :
       END
```

produces exactly the same effect as the more explicit version

```
          FOR A = ∅ $
          FOR B = 1 $
          FOR C = ∅ $
EXECUTE.       BEGIN
               :
               :
                    BEGIN
                    C = C+B $
                    B = B+1 $
                    A = A+1 $
                    IF A LQ 99 $
                         GOTO EXECUTE $
                    END
               END
```

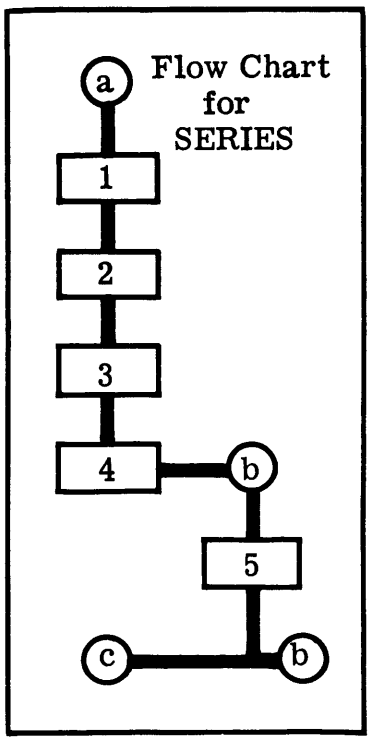
FOR-statement strings are useful in count-controlled loops where non-controlling loop parameters are also counters. A good example of a FOR-statement string is included in the following statement which computes

$$y = \frac{x}{1}\emptyset + \frac{x}{3}1 + \frac{x}{6}2 + \frac{x}{1\emptyset}3 + \cdots = \sum_{i=\emptyset}^{a^{\frac{1}{2}}-1} (x_i \,/\, \sum_{s=1}^{i+1} s)$$

where the subscript A is active and designates the positive value, a.

```
ARRAY XX 1ØØ fixed 36 Signed 15 $''Designates the values of x''
ITEM  YY     fixed 36 Signed 15 $''Designates the value  of y''
        BEGIN SERIES.
  STEP1. YY = XX($Ø$) $
  STEP2. FOR N = A**.5-1 $
  STEP3. FOR I = 1,1,N $
  STEP4. FOR S = 2,S+1 $
  STEP5.       YY = YY+XX($I$)/S $
        END
```

The incomplete, FOR N statement, STEP2, computes the limiting value of I so that it need not be re-computed each pass thru the loop. The flow chart for this statement is shown alongside.



Consider, as another example of FOR-statement strings, the following statement, which computes

$$y = x_\emptyset - \frac{1}{x_1} + x_2 - \frac{1}{x_3} + \cdots - \frac{1}{x_{99}}$$

$$= \sum_{i=\emptyset}^{99} (-1)^i x_i (-1)^i$$

```
ARRAY XX 1ØØ fixed 36 Signed 15 $
ITEM  YY     fixed 36 Signed 15 $
         BEGIN
         YY = XX($Ø$) $
         FOR I = 1,1,99 $
         FOR M = -1,-2*M $
              YY = YY+M*XX($I$)**M $
         END
```

The increment factor in the FOR M statement merely reverses the sign of M each pass thru the loop, so that M EQ (-1)**I.

TEST STATEMENTS. The GOTO statement allows execution control to be transferred to any statement in a program -- except an implicit subscript MODIFY-TEST-REPEAT statement automatically inserted at the bottom of a FOR-loop. Since such a statement is only implied and not written, it can't be named and thus may not be referenced by a GOTO. Nevertheless, the ability to specify a jump from the middle of a loop to a subscript modification at the loop's bottom is often desirable and occasionally necessary. This change in the statement execution sequence is accomplished with a TEST statement, which is therefore defined only within a FOR-loop.

A TEST statement, composed of the TEST sequential operator followed by an optional subscript letter and terminated by the $ separator, completes the current repetition of a FOR-loop by transferring execution control to one of the implicit subscript modifications at the bottom of the loop

statement ∯ TEST ⌊letter⌋ $

A TEST statement without a subscript-letter goes to the first subscript modification of the innermost applicable loop, and thus effects the modification of all the subscripts active at that level. (Recall that the subscript modifications in any given loop are in reverse of the order of their activation.) A TEST statement with a subscript-letter, on the other hand, goes to the modification of the indicated subscript, and may consequently allow some of the subscript modifications to be skipped.

In a relatively simple, one-subscript FOR-loop, the effect of a TEST statement is obvious. Thus,

```
        FOR A = INITIAL,INCREMENT,LIMIT $
             BEGIN
             .
             .
             .
  STEPØ.     TEST $
             .
             .
             .
             END
```

is entirely equivalent to:

```
        FOR A = INITIAL $
EXECUTE.     BEGIN
             .
             .
             .
  STEPØ.     GOTO MODIFY $
             .
             .
             .
                  BEGIN
 MODIFY.          A = A+INCREMENT $
                  IF A GQ;LQ* LIMIT $
                       GOTO EXECUTE $
                  END
             END
```

Scarcely less obvious is its effect in a loop within a loop. For example,

```
        FOR A = INITIALA,INCREMENTA,LIMITA $
             BEGIN
             FOR B = INITIALB, INCREMENTB, LIMITB $
                  BEGIN
                  .
                  .
                  .
  STEPØ.          TEST @ $
                  .
                  .
                  .
                  END
             END
```

has the same effect as:

---

* The choice of GQ or LQ depends on the sign of the increment, as explained on pages 86, 87.

```
                FOR A = INITIALA $
        EXECUTEA.       BEGIN
                        FOR B = INITIALB $
        EXECUTEB.           BEGIN
                            .
                            .
          STEPØ.            GOTO MODIFY@ $
                            .
                            .
                                BEGIN
MODIFY.  MODIFYB.               B = B+INCREMENTB $
                                IF B GQ;LQ* LIMITB $
                                    GOTO EXECUTEB $
                                END
                            END
                            BEGIN
         MODIFYA.           A = A+INCREMENTA $
                            IF A GQ;LQ* LIMITA $
                                GOTO EXECUTEA $
                            END
                    END
```

where @ stands for either A or B. In other words, if the TEST statement in the above example were TEST A $, the effect would be to modify and test subscript A, controlling the outer loop; and if the statement were TEST B $, the effect would be to modify and test subscript B, controlling the inner loop; and finally, if the subscript-letter (signified by @) were omitted entirely, the effect would also be to modify and test subscript B, since it belongs to the innermost loop.

In the remaining case, where the loop is created by a FOR-statement string, the ability of a TEST statement to indicate a particular subscript modification allows a decision to be made within the loop to omit the modification of some of the subscripts (those activated last). To illustrate,

```
        FOR A = INITIALA,INCREMENTA,LIMITA $
        FOR B = INITIALB,INCREMENTB $
        FOR C = INITIALC,INCREMENTC $
            BEGIN
            .
            .
STEPØ.      TEST @ $
            .
            .
            END
```

---

* The choice of GQ or LQ depends on the sign of the increment, as explained on pages 86, 87.

is equivalent to:

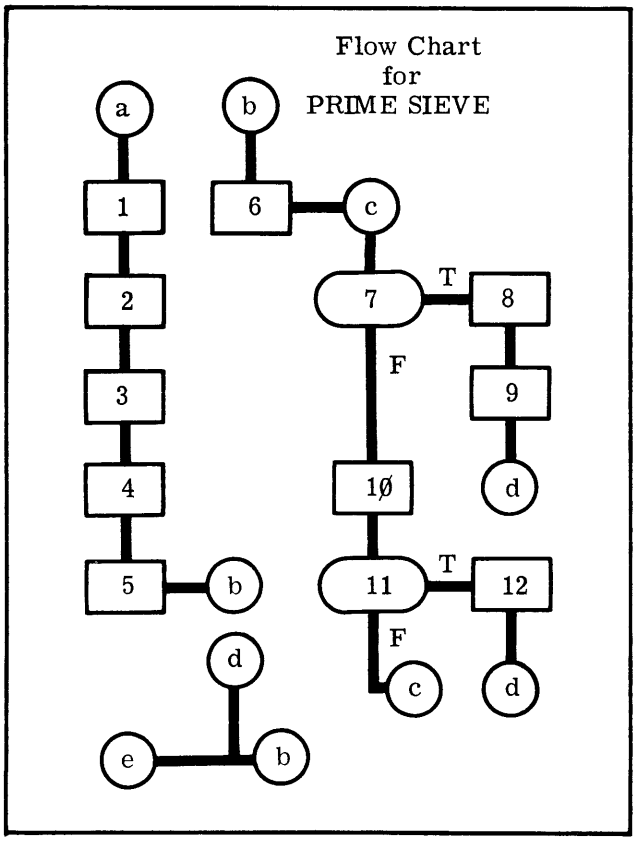
```
                  FOR A = INITIALA $
                  FOR B = INITIALB $
                  FOR C = INITIALC $
        EXECUTE.       BEGIN
                       .
                       .
          STEPØ.       GOTO MODIFY@ $
                       .
                       .
                            BEGIN
MODIFY.  MODIFYC.           C = C+INCREMENTC $
         MODIFYB.           B = B+INCREMENTB $
         MODIFYA.           A = A+INCREMENTA $
                            IF A GQ;LQ LIMITA $
                                 GOTO EXECUTE $
                            END
                       END
```

In the preceding example, a TEST A $ statement skips the modifications of both B and C; a TEST B $ statement skips the modification of C; and a TEST C $ statement causes all three modifications to be performed, as does the simpler but equivalent TEST $ statement.

As an example of the use of the TEST statement, consider the statement PRIME'SIEVE, along with the flow chart illustrating its execution sequence.

```
ARRAY  PRIME 1ØØØØØ fixed 2Ø Unsigned $
ITEM MAXIMUM        fixed 2Ø Unsigned $
PRIME'SIEVE. ''Prime Sieve uses the Sieve of Eratosthenes to
             find all prime numbers not greater than a given
             integer, MAXIMUM, and stores them in an array,
             PRIME.  This version, due to T.C. Wood, appeared
             (in ALGOL) in the March 61 issue of the ACM
             Communications.''
             BEGIN
     STEPØ1. PRIME($Ø$) = 1 $
     STEPØ2. PRIME($1$) = 2 $
     STEPØ3. PRIME($2$) = 3 $
     STEPØ4. FOR N = 3,2,MAXIMUM $
     STEPØ5. FOR J = 2,1 $
                  BEGIN
     STEPØ6.      FOR I = 2,1 $
                       BEGIN  ''Try to factor N''
     STEPØ7.           IF PRIME($I$)**2 GR N $
```

```
                        BEGIN
STEPØ8.                 PRIME($J$) = N $
STEPØ9.                 TEST J $
                        END
STEP1Ø.           FOR K = N/PRIME($I$) $
STEP11.                 IF K EQ N/PRIME($I$) $
STEP12.                     TEST N $
        END  END  END
```



EXERCISE (Loops)

(a) Describe the effect of the following loop.

```
ARRAY GROUP 5Ø Hollerith 6 $
          FOR I = Ø,1,48 $
               GROUP($I$) == GROUP($I+1$) $
```

(b) What effect would the following statement have on the values given after it for the array, LETTER?

```
ARRAY LETTER 5 5 Transmission 1 $
          BEGIN
          FOR I = ∅,1,4 $
               BEGIN
               FOR J = ∅,1,4 $
                    LETTER($I,J$) == LETTER($J,I$) $
               END
          END
```

lT(A)	lT(B)	lT(C)	lT(D)	lT(E)
lT(F)	lT(G)	lT(H)	lT(I)	lT(J)
lT(K)	lT(L)	lT(M)	lT(N)	lT(O)
lT(P)	lT(Q)	lT(R)	lT(S)	lT(T)
lT(U)	lT(V)	lT(W)	lT(X)	lT(Y)

(c) Analyze each of the following loops to determine the number of times the named statement will be executed, assuming it does not affect the operation of the loop.

```
FOR A = 24,-3,∅ $
     BEGIN
     FOR B = ∅,1,A $
          S1.  ---  $
     END
```

```
FOR M = ∅,1,99 $
     BEGIN
     FOR N = ∅,2,M $
          S2.  ---  $
     M = 2*M $
     END
```

```
FOR I = ∅,2,2∅ $
FOR J = I+6,2 $
     BEGIN
     FOR K = ∅,I+1,2*(J-1) $
          S3.  ---  $
     END
```

```
FOR R = ∅,1,number-1 $
FOR S = NUMBER $
     BEGIN
     FOR T = S-1,-1,R $
          S4.  ---  $
     END
```

```
FOR X = 9,2,17 $
     BEGIN
     FOR Y = 1,1,X $
          BEGIN
          FOR Z = 1,Y,1∅∅ $
               BEGIN
               S5.  ---  $
               END
          END
     END
```

(d) Previous statements have activated subscripts M and N, and assigned them values such that (∅ LS M*N LQ 1E4). The statement below "Floogles" the linear array, ELEMENT, an operation that depends on the

parameters M and N. Describe the effect of this statement by defining the term, "Floogle."

```
ARRAY  ELEMENT 1ØØØØ Floating Rounded $
ARRAY FLOOGLED 1ØØØØ Boolean $
ITEM TEMPORARY       Floating Rounded $
      FLOOGLE. BEGIN
               ''Initialize each element to un-floogled.''
        STEP1. FOR K = Ø,1,M*N-1 $
        STEP2.     FLOOGLED($K$) = Ø $
               ''Floogle the array by means of cyclic
               permutations.''
        STEP3. FOR K = Ø,1,M*N-1 $
                     BEGIN
        STEP4.       TEMPORARY = ELEMENT($K$) $
                     ''Floogle this element if not already done.''
        STEP5.       IF NOT FLOOGLED($K$) $
                           BEGIN
        STEP6.             FLOOGLED($K$) = 1 $
                           ''Compute K', floogled index of this
                           element. K' = M*(K modulo(N))+(K/N).''
        STEP7.             FOR P = Ø $
                                 BEGIN
        STEP8.                   IF K GQ N $
        STEP9.                         BEGIN
                                       P = P+1 $  K = K-N $
                                       GOTO STEP8 $
                                       END
       STEP10.                   K = M*K+P $
                                 END
                           ''Exchange elements.''
       STEP11.             TEMPORARY == ELEMENT($K$) $
                           ''Now floogle element just
                           floogled.''
                           GOTO STEP5 $
                           END
                     END
               ''Exchange M and N for floogled array.''
       STEP12. M == N $
               END
```

(e) Compare the following statement with the one on page 98. Make some estimates of their relative efficiency with regard to execution time.

```
ARRAY    WORD 1ØØØ Transmission 5 $
ITEM    WORDS      fixed 1Ø Unsigned Ø...1ØØØ $''Number of words''
ITEM NEW'WORD      Transmission 5 $
      INSERT. ''A statement which inserts a new, 5-letter word
              into an alphabetically ordered list of 5-letter
              words whenever the new word is not already listed.''
              BEGIN
              IF WORDS EQ Ø $
                   BEGIN
                   WORD($Ø$) = NEW'WORD $
                   GOTO UPDATE'WORDS $
                   END
              ''Binary search to determine position of new word.''
              FOR A = Ø $
              FOR Z = WORDS-1 $
      SEARCH. FOR K = (A+Z)/2 $
                   BEGIN
                   IF NEW'WORD NQ WORD($K$) $
                        BEGIN
                        IF NEW'WORD GR WORD($K$) $
                             BEGIN
                             IF A LS Z $
                                  BEGIN
                                  A = K+1 $
                                  GOTO SEARCH $
                                  END
                             K = K+1 $
                             END
                        IF A LS Z $
                             BEGIN
                             Z = K-1 $
                             GOTO SEARCH $
                             END
                        FOR J = WORDS $
                             BEGIN
MOVE'WORD'DOWN.              IF J GR K $
                                  BEGIN
                                  WORD($J$) = WORD($J-1$) $
                                  J = J-1 $
                                  GOTO MOVE'WORD'DOWN $
                                  END
                             END
                        WORD($K$) = NEW'WORD $
 UPDATE'WORDS.          WORDS = WORDS+1 $
              END  END  END
```

(f) Describe the effect of the following statement on the value of the item WORTH in terms of: the array, GAIN; and the next statement, SUCCESS or FAILURE.

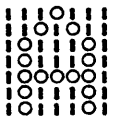
```
ARRAY GAIN 1Ø 2Ø fixed 36 Signed $
ITEM WORTH       fixed 36 Signed $
          BEGIN
          FOR H = Ø $
          FOR I = Ø,1,9 $
               BEGIN
               WORTH = Ø $
               FOR J = Ø,1,19 $
                    BEGIN
                    IF GAIN($I,J$) GR WORTH $
                         BEGIN
                         WORTH = GAIN($I,J$) $
                         H = J $
                         END
                    END
               FOR K = Ø,1,9 $
                    BEGIN
                    IF GAIN($K,H$) LS WORTH $
                         TEST I $
                    END
               GOTO SUCCESS $
               END
          GOTO failure $
          END
```

(g) Revise the example statement on page 90 to allow an empty list, and to accept both positive and negative values for the item, STEP.

(h) Write a JOVIAL statement that lengthens a list of numbers by successively appending sets of differences between adjacent pairs of numbers. Each set of n-1 differences computed from the previous set of n numbers becomes the new set of numbers to be differenced (as long as $n > 2$). Include any necessary declarations. For the numbers a, b, c, and d, the resulting array should be:

```
a
b
c
d
a-b
b-c
c-d
(a-b)-(b-c)
(b-c)-(c-d)
((a-b)-(b-c))-((b-c)-(c-d))
```

(i) Write a JOVIAL statement that determines whether a square array of Boolean values is symmetric along: (1) its horizontal axis; (2) its vertical axis; or (3 & 4) either diagonal axis. Include necessary declarations. The following array, for example,



is symmetric only along its vertical axis.

(j) Write a JOVIAL statement to determine how many sets of numbers, taken in index sequence from an unordered linear array of floating-point numbers, sum to more than 2. but less than 3.

TABLES

A table is a matrix of item values. The rows of a table are called entries, and an entry consists of a related set of different items, perhaps named I1, I2, ---, Im. Typically, entry K, I1(K), I2(K), ---, Im(K), would consist of values measuring the m pertinent attributes of "object" K. Such an entry would be associated with other entries in a table, or list of entries. An n-entry table can be illustrated by the following n by m matrix of subscripted item names:

```
I1($Ø$)     I2($Ø$)     ---  Im($Ø$)
I1($1$)     I2($1$)     ---  Im($1$)
  =           =              =
  =           =              =
I1($K$)     I2($K$)     ---  Im($K$)
  =           =              =
  =           =              =
I1($n-1$)   I2($n-1$)   ---  Im($n-1$)
```

All the entries of a table have the same composition and structure in the sense that each consists of a similarly named and ordered set of items, related to each other by index. The columns of a table are thus linear arrays of index-related items so that, logically, a table is just a collection of such arrays. A particular table item value is designated, as shown above, by item name and entry index. Table items are therefore

processed in much the same manner that linear array items are, except that a loop will usually process an entire entry each pass and not just a single item. However, the advantages of a table over a set of linear arrays make it the favored data structure in JOVIAL programming. The table's advantages arise from the entry concept, which not only explicitly associates related sets of items, but which affords certain processing efficiencies as well, due to the structure of entries and the packing of items within them.

TABLE DECLARATIONS. A table is declared by a table declaration followed by a list of item declarations, enclosed in BEGIN and END brackets, which declare the items comprising a table entry.

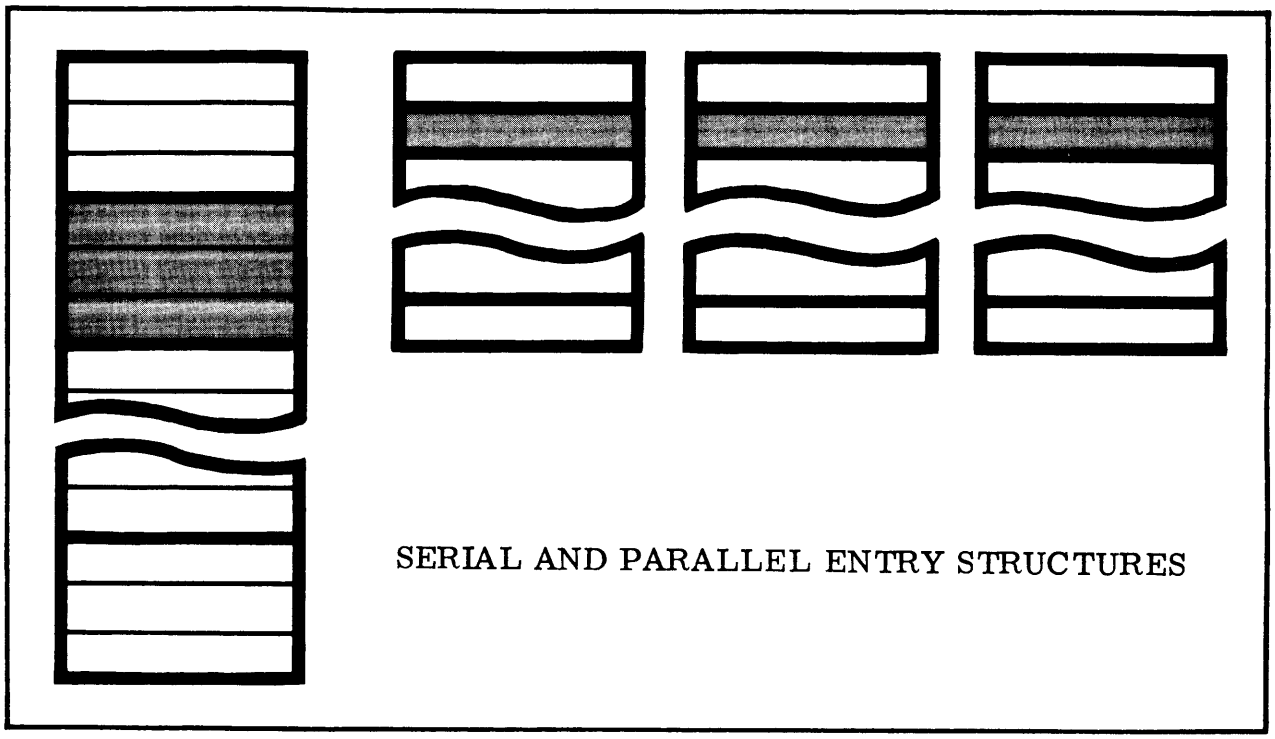
declaration ≠ TABLE ⌊name$_{\text{of-table}}$⌋ Variable;Rigid$_{\text{length}}$ number$_{\text{of-entries}}$

⌊Serial;Parallel$_{\text{entry-structure}}$⌋ ⌊No;Medium;Dense$_{\text{item-packing}}$⌋ $ BEGIN

⌈ITEM name$_{\text{of-table-item}}$ description $⌉s END

A table name may be omitted from the declaration if only individual table items are referred to in the program, and never the entire table. The Variable or Rigid length descriptors determine whether the number of entries will be allowed to vary during the execution of the program, and for a Variable length table, number of entries indicates the table's maximum length. The Serial or Parallel entry structure descriptors allow the programmer, if he desires, to indicate one of two possible storage configurations for the table: Serial entry structure means that entries are allocated serial, or consecutive, blocks of storage space; while Parallel entry structure means that the table is divided into separate blocks, and entries are allocated parallel, or similarly located, registers within them.

The distinction between Serial and Parallel entry structure is graphically illustrated below:

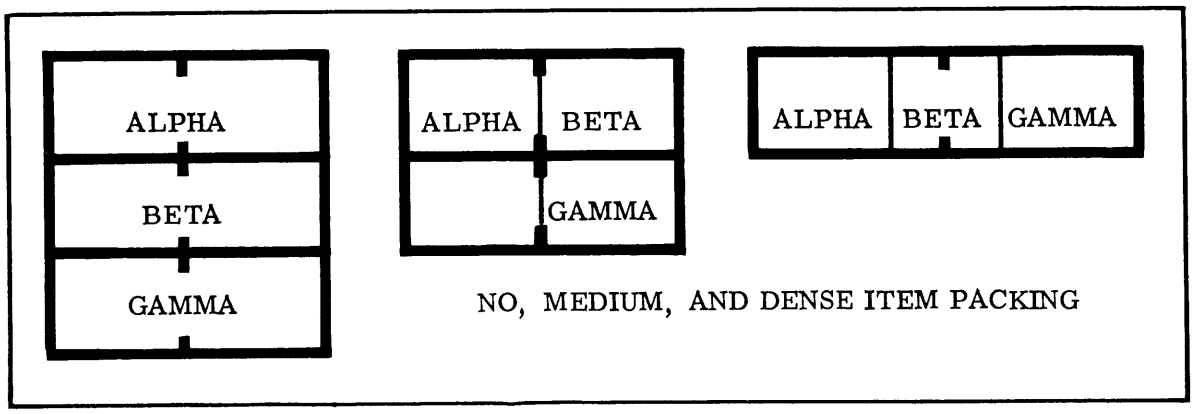


The No, Medium, or Dense item packing descriptors allow the programmer, again if he desires, to indicate one of three possible storage allocation schemes for the items in an entry: No packing means that storage is allocated in full register units, so that each item in the entry will occupy one or more consecutive computer words; Medium packing means that storage is allocated in sub-register* units, so that each item in the entry will occupy one or more consecutive sub-words; and Dense packing means that storage is allocated in bit position units, so that each item in the entry will occupy one or more consecutive bit positions.

---

*Many computers have instructions that, by effectively partitioning memory registers into two or more segments, greatly facilitate extracting values from or inserting them into these segments. Whether left-half-word and right-half-word or prefix, decrement, tag, and address, these natural segments are called sub-registers.

The three item packing schemes are shown in the following diagram:



While No item packing may save the most execution time, Dense item packing will usually save the most storage space, and Medium item packing will usually afford intermediate savings in both time and space.

The programmer need indicate neither entry structure nor item packing for a table, and if either descriptor is omitted, the compiler will supply its standard description.

To illustrate the table concept and to exemplify the TABLE declaration, consider the following three tables, which contain information on employees, airbases, and air navigation beacons.

```
TABLE PAYROLL Variable 1ØØØ $
              BEGIN
ITEM EMP'NAME Hollerith 18 $
ITEM MAN'NMBR fixed 12 Unsigned $
ITEM ORG'CODE Status V(SALES) V(PROD) V(ENG) V(RES) V(PERS) $
ITEM PAY'RATE fixed 1Ø Unsigned $
ITEM JOB'TIME fixed Ø8 Unsigned 2 $
ITEM NET'EARN fixed 2Ø Unsigned $
ITEM YTD'EARN fixed 24 Unsigned $
              END
```

```
TABLE AIRBASE'WEATHER Rigid 8ø Serial Dense $
                      BEGIN
ITEM     AIRBASE'CODE Hollerith 3''letters''$
ITEM      REPORT'HOUR fixed 5 Unsigned ø...23''hours''$
ITEM    REPORT'MINUTE fixed 6 Unsigned ø...59''minutes''$
ITEM   WEATHER'CHANGE Boolean $
ITEM  CURRENT'SUMMARY Status V(OPEN) V(INSTRUMENT) V(CLOSED) $
ITEM FORECAST'SUMMARY Status V(OPEN) V(INSTRUMENT) V(CLOSED) $
ITEM          CEILING fixed 9 Unsigned ø...511''hundred feet.  Maximum
                      of 511 means unlimited''$
ITEM       VISIBILITY fixed 5 Unsigned 1 ø...15.5Al''nautical miles.
                      Maximum of 15.5 means unlimited''$
ITEM VISIBILITY'BLOCK Status V(NONE) V(FOG) V(DUST) V(SMOKE) V(HAZE) $
ITEM     BLOCK'AMOUNT Status V(LIGHT) V(MODERATE) V(HEAVY) $
ITEM    PRECIPITATION Status V(NONE) V(RAIN) V(SNOW) V(SLEET) V(HAIL) $
ITEM    PRECIP'AMOUNT Status V(LIGHT) V(MODERATE) V(HEAVY) $
ITEM RUNWAY'CONDITION Status V(OK) V(WET) V(ICY) V(SNOW) V(BLOCKED) $
                      END


TABLE       AIRWAY'FIXES Variable 8øøø Medium $
                         BEGIN
ITEM LOCATION'IDENTIFIER Hollerith 3''letter teletype abbreviation''$
ITEM  CONTROLLING'AGENCY Status V(ATLANTA) V(BOSTON) V(CHICAGO)
                         V(CLEVELAND) V(DETROIT) V(FT'WORTH) V(GARDINER)
                         V(INDIANAPOLIS) V(KANSAS'CITY) V(MEMPHIS)
                         V(MIAMI) V(MINNEAPOLIS) V(MONTREAL) V(MONCTON)
                         V(NEW'ORLEANS) V(NEW'YORK) V(NORFOLK)
                         V(OCEANIC'NEW'YORK) V(PITTSBURGH) V(ST'LOUIS)
                         V(TORONTO) V(WASHINGTON) $
ITEM            LOCATION Dual 16 Unsigned 5''in nautical miles East,North
                         of sector center''$
ITEM  MAGNETIC'VARIATION fixed 16 Signed''in seconds of arc--East=plus,
                         West=minus''$
ITEM           REPORTING Status V(UNNECESSARY) V(COMPULSORY) V(OPTIONAL) $
ITEM            FACILITY Status V(VORTAC) V(VOR) V(RANGE'STATION)
                         V(RADIO'BEACON) V(AIRPORT'AIRBASE)
                         V(REPORTING'POINT) V(AIRWAY'INTERSECTION) $
ITEM              SECTOR Status V(EXTERNAL) V(NORTH) V(SOUTH) V(EAST)
                         V(WEST) $
ITEM        IN'OPERATION Boolean $
                         END
```

It is even possible to have a table of information about tables!

```
TABLE      CTL ''COMPOOL Table List''Variable 5ØØ Serial Dense $
               BEGIN
ITEM       TAG Hollerith 8''characters maximum,right justified''$
ITEM      TYPE Status V(VARIABLE'LENGTH) V(RIGID'LENGTH) $
ITEM    LENGTH fixed 16 Unsigned''in entries''$
ITEM   STORAGE fixed 16 Unsigned''in registers''$
ITEM      FORM Status V(PARALLEL) V(SERIAL) $
ITEM   PACKING Status V(NONE) V(MEDIUM) V(DENSE) $
ITEM ITEM'ONE fixed 16 Unsigned''CIL (COMPOOL Item List) entry index
               for 1st item of table, which itself contains the entry
               index of the 2nd item, etc.''$
ITEM  IN'CORE Boolean $
ITEM  ON'FILE Boolean $
ITEM  ADDRESS ''If IN'CORE''fixed 16 Unsigned $
ITEM FILE'TAG ''If ON'FILE''Hollerith 8''characters maximum,right
               justified''$
ITEM LOCATION ''If ON'FILE''fixed 16 Unsigned''position index of
               record''$
               END
```

LIKE TABLE DECLARATIONS. In some cases, a program's environment must contain two or more instances of tables with the same entry structure. Assuming that one of the tables is already declared, either in the COMPOOL or in the program, it is tedious to have to declare a new but essentially similar table completely -- especially one with many items. Such tables may therefore be declared and named, using a previously defined table as a pattern, by adding a distinguishing letter or numeral to the pattern table's name.

declaration ≠ TABLE name$_{\text{of-pattern-table}}$:letter;numeral ⌊Variable;

Rigid$_{\text{length}}$ number$_{\text{of-entries}}$⌋ ⌊Serial;Parallel$_{\text{entry-structure}}$⌋ ⌊No;Medium;

Dense$_{\text{item-packing}}$⌋ Like $

The like table may have its own descriptions of length, entry structure, and item packing declared, or it may retain those of the pattern table. No list of item declarations is necessary after a like table declaration, for the composition and structure of the like table's entries are taken as being generated by the declarations describing the pattern table's entries, with the exception that all item names are suffixed with the distinguishing letter or numeral. Thus, the declaration

```
TABLE  CTLØ  Like''CTL declared above''$
```

automatically declares the table items TAGØ, TYPEØ, LENGTHØ, and so on. Care must be taken in choosing a distinguishing letter or numeral to ensure that item names resulting from a like table declaration are unique, and do not accidentally conflict with other identifiers.

The following declarations show some of the ways in which a like table's description may differ from that of the pattern table.

```
TABLE  PAYROLLA                          No    Like $

TABLE  PAYROLLB  Rigid      1 Serial    Dense Like $

TABLE  PAYROLLC  Variable 1ØØ Parallel        Like $
```

FUNCTIONAL MODIFIERS

JOVIAL's functional modifiers are, in a sense, extensions to the basic language, which is essentially an item manipulating language. They allow the programmer to conveniently describe the manipulation of both larger data structures than items (i.e., entries and tables) and smaller data structures (i.e., segments of the machine symbols representing item values). Functional modifiers have the general form of functions, modifying a table or table item name, or an item value.

TABLE MANIPULATING FUNCTIONAL MODIFIERS. Tables are the important data structures in most JOVIAL programs, so the language provides several functional modifiers to aid in their manipulation.

NENT. A vital parameter in table processing is number of entries. The functional modifier NENT allows this unsigned, integral value to be designated for variable length tables, and specified for rigid length tables.

variable$_{\text{of-numeric-type}}$ ≠ N$_{\text{umber-of}}$:ENT$_{\text{ries}}$ (name$_{\text{of-variable-length-table-or-table-item}}$)

numeric-formula ≠ N$_{\text{umber-of}}$:ENT$_{\text{ries}}$ (name$_{\text{of-rigid-length-table-or-table-item}}$)

NENT performs two valuable services: for variable length tables, it conveniently expresses current number of entries; and for rigid length tables, it insulates from change those statements that refer to number of entries. For variable length tables, NENT acts as a counter that the program itself must update whenever it changes the table's length. The following statement, for example, records the addition of a new entry to the CTL table.

```
NENT(CTL) = NENT(CTL)+1 $
```

For rigid length tables, NENT acts as a preset parameter. Thus, when a redesign changes the length of a fixed table, the new value for number of entries is automatically compiled into the program wherever it is specified by NENT.

NWDSEN. Another parameter in table processing is the amount of storage allocated to a table entry (and thus to the entire table). This unsigned, integral value, which is constant throughout the execution of the program is expressed in number of words, or registers, per entry and may be specified with the functional modifier NWDSEN.

numeric-formula ¢ $N_{umber\text{-}of}:W_{or}:DS_{per}:EN_{try}$ ($name_{of\text{-}table\text{-}or\text{-}table\text{-}item}$)

Although number of words per entry is almost never used in ordinary JOVIAL programming, its use is necessary in executive programs that perform dynamic storage allocation*. For example:

```
CTL'SIZE'IN'WORDS = NWDSEN(CTL)*NENT(CTL) $
```

ALL. A very common loop in JOVIAL programming cycles through an entire table, processing one entry each repetition, with the number of passes equal to the number of entries. While such a loop (for the CTL table, for example) can be created by either

```
FOR T = Ø,1,NENT(CTL)-1 $
```

which processes down from the top of the table, or

---

*Storage allocation at time of program execution rather than program compilation.

FOR T = NENT(CTL)-1,-1,Ø $

which processes up from the bottom, it is shorter and much more descriptive to use the functional modifier ALL in the abbreviated form of the complete FOR statement.

statement ↞ FOR letter = ALL ($name_{\text{of-table-or-table-item}}$) $

For example:

FOR T = ALL (CTL) $

which is, in effect, an abbreviation of one of the previous two statements. (Just which of these two statements the FOR - ALL statement abbreviates is not defined, so that its usefulness is limited to those loops where direction of processing is unimportant. The dependency of the correct functioning of a loop on its direction of processing is often quite subtle, however, especially where the loop itself affects the number of entries in the (variable length) table, so that reasonable caution is necessary.

ENTRY. As mentioned before, a table entry is a conglomeration of related items. The functional modifier ENTRY allows an entry to be considered as a single value, represented by a single, composite symbol. An entry's value may be denoted by Ø if all its items have values represented by zero; otherwise, its value is not denotable. Entry values may be compared (for equality/inequality), assigned, and exchanged.

entry-variable ↞ ENTRY ($name_{\text{of-table-or-table-item}}$ ($ $index_{\text{of-entry}}$ $))

boolean formula ↞ entry-variable EQ;NQ Ø;entry-variable

statement ↞ entry-variable = Ø;entry-variable $

statement ↞ entry-variable == entry-variable $

The comparing, assigning, and exchanging of entry values operate as if on unsigned integers, although this is not of interest unless entries of different size are involved. In such cases, the shorter entry is effectively prefixed by registers containing zero.

The following statement, which eliminates empty or zero entries from the previously declared PAYROLL table, illustrates the use of the ENTRY modifier, and the NENT and ALL modifiers as well.

```
            BEGIN
            FOR I = ALL (PAYROLL) $
                 BEGIN
SEEK'EMPTY.      IF ENTRY (PAYROLL($I$)) EQ Ø $
                      BEGIN
                      NENT(PAYROLL) = NENT(PAYROLL)-1 $
                      IF I LS NENT(PAYROLL) $
                           BEGIN
                           ENTRY (PAYROLL($NENT(PAYROLL)$)) == ENTRY
(PAYROLL($I$)) $           GOTO SEEK'EMPTY $
            END  END  END  END
```

EXERCISE (Tables)

(a) Declare an inventory table with the following information: part number; part name; amount on hand; unit price; unit cost; gross sales to date; reorder point; and reorder quantity.

(b) Declare a transaction table with the following information: part number; quantity; shipment or receipt.

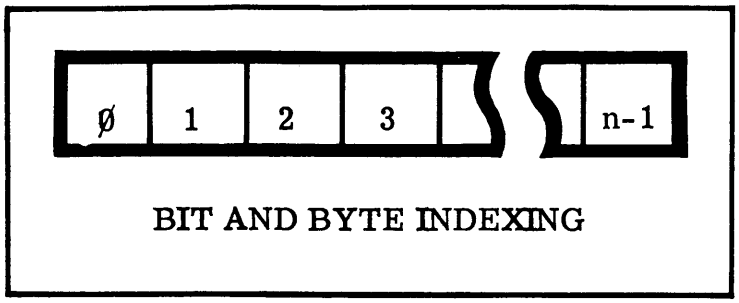
(c) Write a JOVIAL statement to update the inventory table from the information in the transaction table.

(d) Declare a table containing the information pertaining to a list of machine instructions for any particular computer.

(e) Declare a table containing the information to be found on a driver's license. Assume that storage space must be conserved.

SYMBOL MANIPULATING FUNCTIONAL MODIFIERS. Although the item is normally the smallest unit of data in JOVIAL, it is occasionally necessary to designate a value represented by part of an item's machine symbol. This is especially true of literal items, which must often be considered as linear arrays of individual signs.

BIT AND BYTE. The machine symbol representing any item's value may be considered a string of bits or, in the case of literal items, of 6-bit bytes. In either case, both bits and bytes are indexed, left to right, from Ø to n-1 as shown below for an n-element symbol.



BIT AND BYTE INDEXING

The BIT modifier allows any segment of the bit string representing the value of any item to be designated as an unsigned, integral variable. And, in a similar fashion, the BYTE modifier allows any segment of the byte string representing the value of any literal item to be designated as a literal variable. The first bit or byte of the segment and the number of bits or bytes in the segment are specified by the 2-component index, enclosed in the subscript brackets ($ and $), after the modifier.

variable$_{\text{of-numeric-type}}$ ≠ BIT ($ numeric-formula$_{\text{index-of-first-bit}}$

⌊, numeric-formula$_{\text{number-of-bits}}$⌋ $) (name$_{\text{of-item}}$ ⌊($ index $)⌋)

variable$_{\text{of-literal-type}}$ ≠ BYTE ($ numeric-formula$_{\text{index-of-first-byte}}$

⌊, numeric-formula$_{\text{number-of-bytes}}$⌋ $) (name$_{\text{of-literal-item}}$ ⌊($ index $)⌋)

If a segment of length one is desired, the numeric formula specifying number of bits or bytes may be omitted from the index subscripting the modifier. Thus,

```
BIT($I$)(EMP'CODE)
```

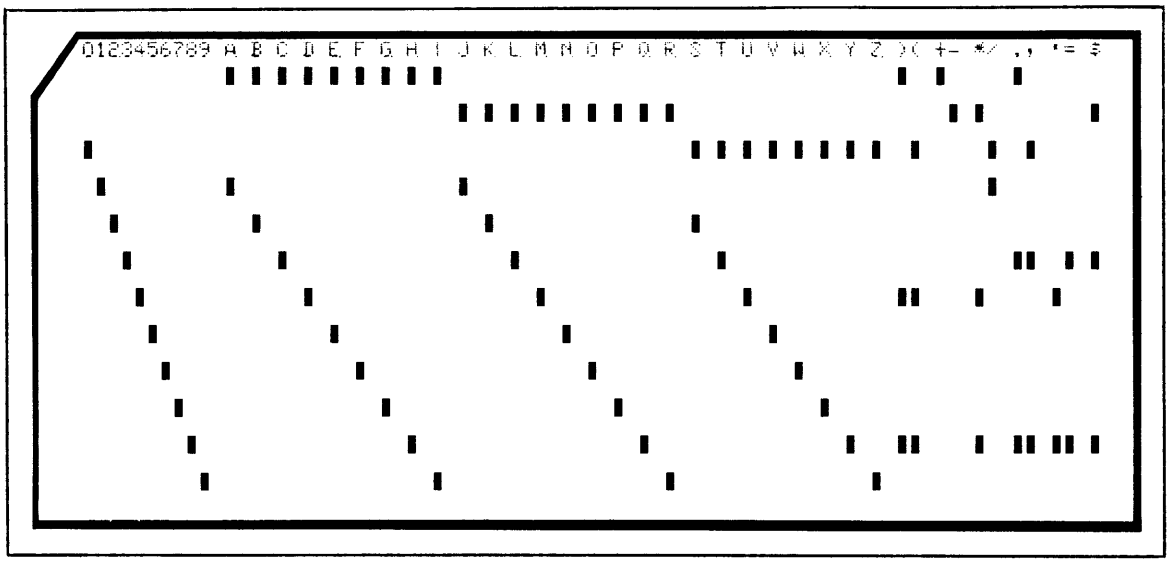
and

```
BIT($I,1$)(EMP'CODE)
```

both specify the same 1-bit integer. A trio of more elaborate examples illustrate the utility of the BIT and BYTE modifiers for symbol manipulation.

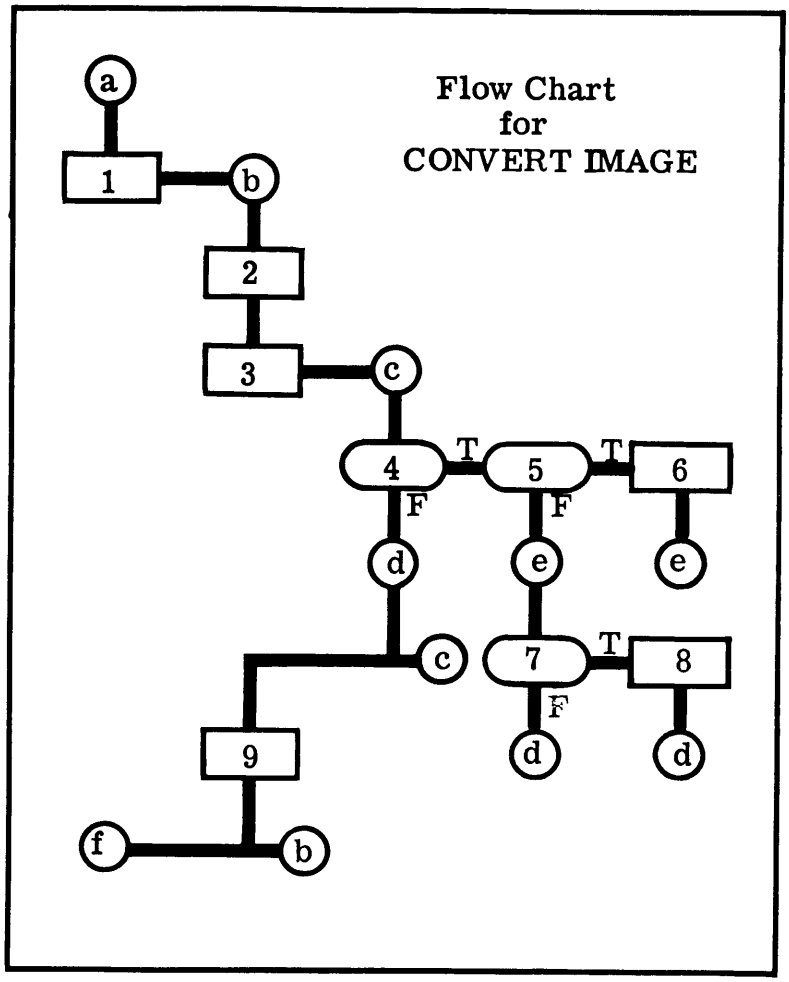
Notice how the BIT modifier is used to designate the numeric encoding of non-numeric values.

The first routine converts a 12-row by 8ø-column punched card image, as shown below, into an 8ø-character literal value, in the Hollerith coding of a particular computer.




```
     CONVERT'IMAGE. BEGIN''A routine to convert from a punched card image
                    to an 8ø-character, Hollerith-coded, literal value.
                    Illegal punch combinations are not rejected and may
                    cause spurious results.''
ARRAY          PUNCH 12 8ø Boolean $
ITEM            CARD Hollerith 8ø $
ITEM          COLUMN Hollerith ø1 $
              STEP1. FOR J = ø,1,79 $
                           BEGIN
              STEP2.       COLUMN = 0(øø) $
              STEP3.       FOR I = ø,1,11 $
                                 BEGIN
              STEP4.             IF PUNCH($I,J$) $
                                       BEGIN
            STEP5'6.                   IF I LQ 2 $  BIT($ø,6$)(COLUMN) =
BIT($ø,6$)(COLUMN)+(I+1)*0(2ø) $
            STEP7'8.                   IF I GR 2 $  BIT($ø,6$)(COLUMN) =
BIT($ø,6$)(COLUMN)+(I-2) $    END  END
              STEP9.       BYTE($J$)(CARD) = COLUMN $
                           END  END
```

The statement execution flow chart for this routine follows:



The second routine analyzes an arbitrary symbol and places it in a string of labels if it happens to be a grammatical JOVIAL label.

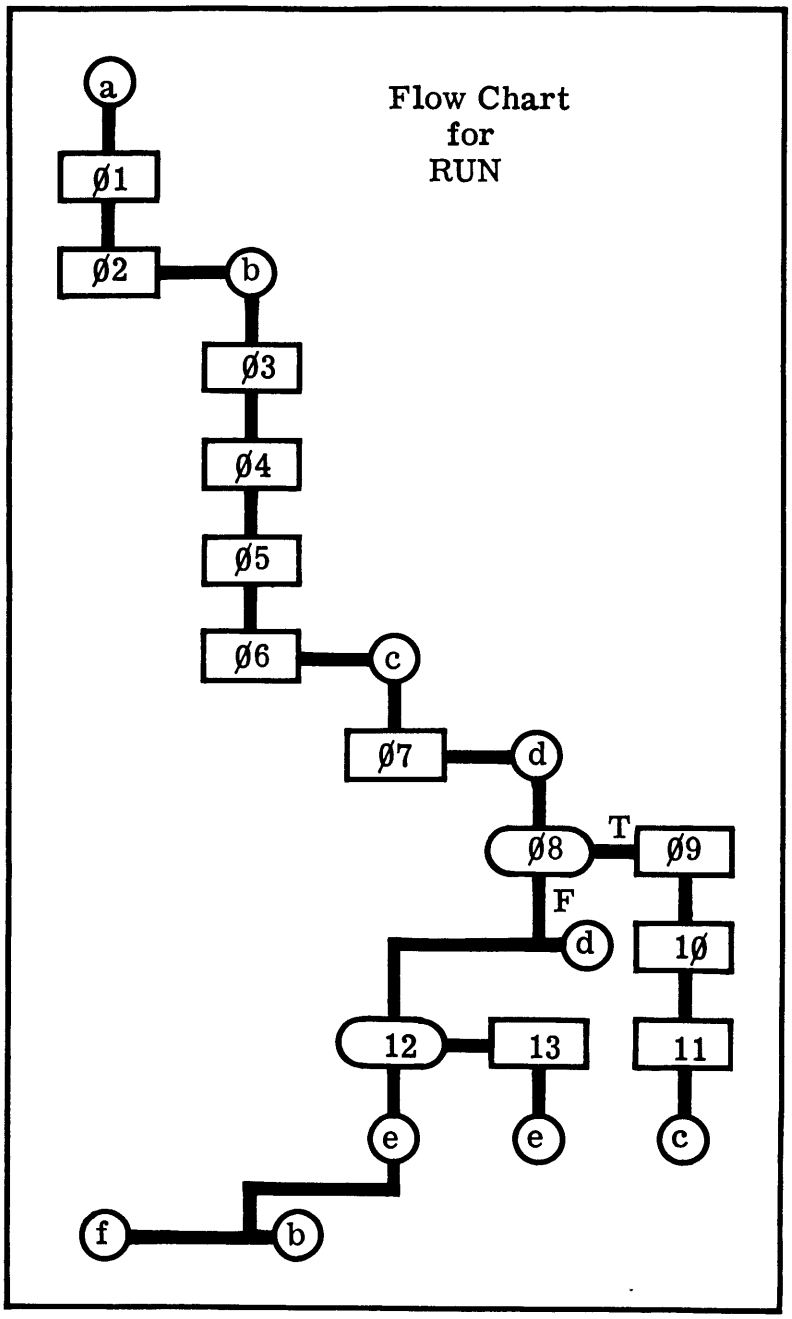
```
    PROCESS'SYMBOL. BEGIN''A routine to determine whether an arbitrary
                    symbol is a JOVIAL label and, if so, place it (in
                    the sense of either find or file) in a label string,
                    setting the slot to the index of the dictionary
                    entry describing the label.''
ITEM          LABEL Transmission 1ØØØØ''characters''$
ITEM         SYMBOL Transmission 64''characters, right justified''$
ITEM    SYMBOL'TYPE Status V(NON'LABEL) V(OLD'LABEL) V(NEW'LABEL)
                    V(EXCESS'LABEL) $
ITEM           SLOT fixed 11 Unsigned''dictionary index''$
TABLE   LABEL'INDEX Variable 2ØØØ''entries''$
                    BEGIN
ITEM         ORIGIN fixed 14 Unsigned''character index''$
ITEM         LENGTH fixed Ø6 Unsigned''number of characters''$
                    END

                    FOR I = 64,-1 $
                         BEGIN
      FIND'ORIGIN.       IF I EQ Ø OR BYTE($I-1$)(SYMBOL) EQ 1T( ) $
                              BEGIN
                              IF I LS 63 AND 1T(A) LQ BYTE($I$)(SYMBOL)
LQ 1T(Z) $                         BEGIN
                                   FOR J = I+1,1,63 $
                                        BEGIN
       LABEL'TEST.                      IF NOT (1T(A) LQ BYTE($J$)
(SYMBOL) LQ 1T(Z) OR 1T(Ø) LQ BYTE($J$)(SYMBOL) LQ 1T(9) OR (BYTE($J$)
(SYMBOL) EQ 1T(') AND J NQ 63 AND BYTE($J+1$)(SYMBOL) NQ 1T('))) $
                                             GOTO NON'LABEL $
                                        END
                                   FOR K = 64-I $
      PLACE'LABEL.                      BEGIN
       FIND'LABEL.                      FOR L = ALL (LABEL'INDEX) $
                                             BEGIN
                                             IF K EQ LENGTH($L$) AND
BYTE($I,K$)(SYMBOL) EQ BYTE($ORIGIN($L$),LENGTH($L$)$)(LABEL) $
                                                  BEGIN
                                                  SLOT = L $
                                                  SYMBOL'TYPE =
V(OLD'LABEL) $                                    GOTO EXIT $
                                             END  END
```

```
        FILE'LABEL.                          FOR M = ORIGIN($NENT(INDEX'LABEL)
- 1$) + LENGTH($NENT(INDEX'LABEL)-1$) $            BEGIN
     TEST'CAPACITY.                               IF NENT(LABEL'INDEX) EQ 2ØØØ
OR M+K GQ 1ØØØØ $                                      BEGIN
                                                       SYMBOL'TYPE =
V(EXCESS'LABEL) $                                      GOTO EXIT $
                                                       END
                                                  SLOT = NENT(LABEL'INDEX) $
                                                  SYMBOL'TYPE = V(NEW'LABEL) $
                                                  BYTE($M,K$)(LABEL) = BYTE
($I,K$)(SYMBOL) $                                 ORIGIN($SLOT$) = M $
                                                  LENGTH($SLOT$) = K $
                                                  NENT(LABEL'INDEX) = NENT
(LABEL'INDEX)+1 $                            END  END
                                       GOTO EXIT $
                                       END
          NON'LABEL.                SYMBOL'TYPE = V(NON'LABEL) $
                                    GOTO EXIT $
                    END  END  END
               EXIT.
```

And the third routine computes the length of the longest run in a bridge hand, in thirteen steps.

```
              RUN. BEGIN''A routine to compute the length of the longest
                   run or unbroken sequence of cards of the same suit in
                   a bridge hand of 13 playing cards. (The ace is either
                   high or low.)''
TABLE         CARDS Rigid 13 $
                   BEGIN
ITEM      CARD'SUIT Status 6 V(CLUB) V(DIAMOND) V(HEART) V(SPADE) $
ITEM      CARD'NAME Status 6 V(DEUCE) V(TREY) V(FOUR) V(FIVE) V(SIX)
                   V(SEVEN) V(EIGHT) V(NINE) V(TEN) V(JACK) V(QUEEN)
                   V(KING) V(ACE) $
                   END
ITEM     RUN'LENGTH fixed 4 Unsigned $
ITEM           SUIT Status 6 V(CLUB) V(DIAMOND) V(HEART) V(SPADE) $
ITEM           NAME Status 6 V(DEUCE) V(TREY) V(FOUR) V(FIVE) V(SIX)
                   V(SEVEN) V(EIGHT) V(NINE) V(TEN) V(JACK) V(QUEEN)
                   V(KING) V(ACE) $
ITEM     FIRST'CARD Boolean $
           STEPØ1. RUN'LENGTH = Ø $
           STEPØ2. FOR I = ALL (CARDS) $
                        BEGIN
           STEPØ3.      FIRST'CARD = 1 $
           STEPØ4.      SUIT = CARD'SUIT($I$) $
           STEPØ5.      NAME = CARD'NAME($I$) $
           STEPØ6.      FOR J = 1 $
   FIND'NEXT'CARD.           BEGIN
           STEPØ7.           FOR K = ALL (CARDS) $
                                  BEGIN
           STEPØ8.                IF CARD'SUIT($K$) EQ SUIT AND (BIT
($Ø,6$)(CARD'NAME($K$)) EQ BIT($Ø,6$)(NAME)+1 OR (FIRST'CARD AND NAME
EQ V(ACE) AND CARD'NAME($K$) EQ V(DEUCE))) $
                                       BEGIN
           STEPØ9.                     NAME = CARD'NAME($K$) $
           STEP1Ø.                     J = J+1 $
           STEP11.                     FIRST'CARD = Ø $
                                       GOTO FIND'NEXT'CARD $
                                  END  END
           STEP12.           IF J GR RUN'LENGTH $
           STEP13.                RUN'LENGTH = J $
                   END  END  END
```


Flow Chart
for
RUN
a
Ø1
Ø2
b
Ø3
Ø4
Ø5
Ø6
c
Ø7
d
Ø8
T
Ø9
F
d
1Ø
12
13
11
e
e
c
f
b

MANT AND CHAR. A floating-point machine symbol representing a numeric value consists of: a mantissa, which is a signed fraction representing the significant digits of the value; and a characteristic, which is a signed integer representing the exponent of an implicit power of two scaling factor for the mantissa. Either component of a simple or subscripted floating-point item may be designated as fixed-point variables; the mantissa with the functional modifier MANT, and the characteristic with the functional modifier CHAR.

variable$_{\text{of-numeric-type}}$ ≠ MANT;CHAR (name$_{\text{of-floating-item}}$ ⌊($ index $)⌋)

Thus, the fixed-point value of the floating-point item ALPHA(I) can be specified as:

```
MANT(ALPHA($I$))*2**CHAR(ALPHA($I$))
```

and multiplication of a floating-point item by a power of two (e.g., 2**J) can be stated as:

```
CHAR(ALPHA($I$)) = CHAR(ALPHA($I$))+J $
```

and whether two floating-point items are in the same range of magnitude can be determined with the relational Boolean formula:

```
CHAR(ALPHA($I$)) EQ CHAR(ALPHA($I+1$))
```

ODD. In numeric computations, it is occasionally necessary to determine whether the least significant bit of the machine symbol representing the value of a subscript or a numeric item actually represents the value one, or zero; for integers, in other words, whether the value is odd or even. For subscripts or for simple or subscripted numeric (i.e., fixed or floating-point) items, this Boolean value, odd or even, may be designated as a Boolean variable with the functional modifier ODD, which, insulated from both the length and coding of the machine symbol it affects, designates True if the least significant bit of that symbol represents a magnitude of one, and False if it represents a magnitude of zero.

variable$_{\text{of-boolean-type}}$ ≠ ODD (letter; ⌈name$_{\text{of-fixed-or-floating-item}}$ ⌊($ index $)⌋⌉)

Notice that, because of differences between computers in encoding negative numeric values,

ODD(BETA) AND BIT(5)(BETA)

is not always True for a fixed-point, 6-bit, signed item, BETA, for example. The functional modifier ODD, operating on a loop controlling subscript, can be used as an alternator, which is a Boolean formula within a loop that only specifies True every other pass. Consider the ODD alternator in the following statement, which counts (1) all odd integers and (2) all oddly indexed integers in an array of floating point numbers.

```
ITEM COUNT'1       fixed 9 Unsigned''count of odd integers in ALPHA''$
ITEM COUNT'2       fixed 9 Unsigned''count of oddly indexed integers in ALPHA''$
ARRAY  ALPHA 1ØØØ Floating Rounded $
                   COUNT'1 = Ø $  COUNT'2 = Ø $
                   FOR I = Ø,1,999 $
                        BEGIN
                        IF CHAR(ALPHA($I$)) EQ MANTISSA'LENGTH''in bits''$
                             BEGIN
                             IF ODD(I) $  COUNT'2 = COUNT'2+1 $
                             IF ODD(ALPHA($I$)) $  COUNT'1+1 $
                        END  END
```

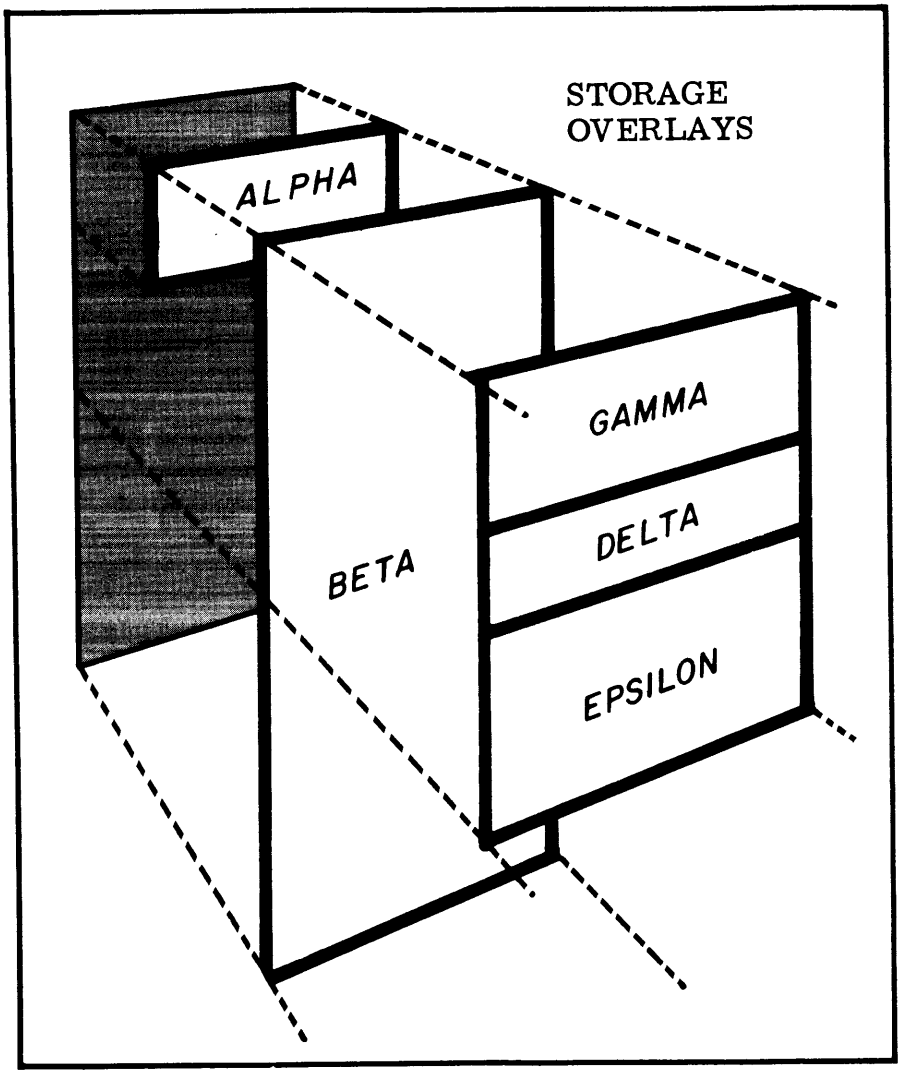
MISCELLANEOUS DECLARATIONS

OVERLAY DECLARATIONS. An OVERLAY declaration serves to arrange previously declared items, arrays, and tables in memory by allocating blocks of storage space to them. The declaration itself is composed of one or more lists of item, array item, and table names, separated from each other by the = separator and enclosed by the OVERLAY declarator and the terminating $ separator. The individual names in each list are separated by the , separator and are allocated, in sequence, consecutive units of storage space from a common block of storage sufficient to contain the largest set of data elements listed in the declaration. Data elements in one list consequently "overlay" those in the other lists of the declaration.

declaration ≠ OVERLAY ⌈names'$_{\text{of-items-arrays-tables}}$⌉s$^{=}$ $

The following declaration, for example, might produce the storage overlays illustrated below.

```
OVERLAY ALPHA=BETA=GAMMA,DELTA,EPSILON $
```



An OVERLAY declaration, appearing inside the BEGIN and END brackets of a TABLE declaration, can be used to overlay table items within the table's entries, but may list only those item names previously declared as part of the table. For example, the following declaration could be used to declare a Hollerith input table whose items are to be converted to more convenient representations for easier manipulation.

```
TABLE   PAYROLL Variable 1ØØØ $
                BEGIN
ITEM H'EMP'NAME Hollerith 18 $
ITEM   EMP'NAME Transmission 18 $
OVERLAY         H'EMP'NAME = EMP'NAME $
ITEM H'MAN'NMBR Hollerith 4 $
ITEM   MAN'NMBR fixed 12 Unsigned $
OVERLAY         H'MAN'NMBR = MAN'NMBR $
ITEM H'ORG'CODE Hollerith 5 $
ITEM   ORG'CODE Status V(SALES) V(PROD) V(ENG) V(RES) V(PERS) $
OVERLAY         H'ORG'CODE = ORG'CODE $
ITEM H'PAY'RATE Hollerith 5 $
ITEM   PAY'RATE fixed 1Ø Unsigned $
OVERLAY         H'PAY'RATE = PAY'RATE $
ITEM H'JOB'TIME Hollerith 5 $
ITEM   JOB'TIME fixed 8 Unsigned 2 $
OVERLAY         H'JOB'TIME = JOB'TIME $
ITEM H'NET'EARN Hollerith 7 $
ITEM   NET'EARN fixed 2Ø Unsigned $
OVERLAY         H'NET'EARN = NET'EARN $
ITEM H'YTD'EARN Hollerith 9 $
ITEM   YTD'EARN fixed 24 Unsigned $
OVERLAY         H'YTD'EARN = YTD'EARN $
                END
```

To save storage space, the converted items in the above table overlay their Hollerith equivalents, under the assumption that, after conversion, the Hollerith values are no longer needed.

As the result of an OVERLAY declaration, storage space is allocated in blocks of consecutive units to the data elements named in the declaration. For arrays and tables, these units are full memory registers; and for items, the units are registers, sub-registers, or bit positions -- depending on whether the item-packing mode is No packing, Medium packing, or Dense packing. Each data element named in the declaration is thereby allocated a block of consecutive registers, of consecutive sub-registers, or of consecutive bit positions. Data elements whose names are preceded in the declaration by either the OVERLAY declarator or the = separator are allocated storage beginning at an unspecified origin, while elements whose names are preceded by the , separator are allocated storage immediately after the block allocated the previously named element (except

within Medium and Densely packed tables, where the declared item sequence may occasionally be altered to conform to sub-register partitions). To take maximum advantage of whatever natural memory partitioning may exist in a particular computer, data elements may be allocated more storage than they actually require and, therefore, need not completely fill up the storage allocated to them. Thus, even with Dense item packing, a less-than-register-size table item might be allocated an entire register, to avoid other items in the entry being unnecessarily split between registers. It is, therefore, often difficult to exactly determine how many units of storage will be allocated any given data element, and this in turn creates difficulties in declaring precise overlays.

An OVERLAY declaration can be used simply to declare a storage sequence for a single set of data elements in order, for instance, to form effective groupings for input/output purposes, or to conform to some pre-established arrangement. For example, the following declaration,

```
OVERLAY HEAD,BODY,TAIL $
```

arranges the three data elements one after the other in storage, but does not create any overlays.

The main function of the OVERLAY declaration, however, is that of creating overlays -- by allowing multiple descriptions of the structure and coding of a single block of storage. The main purpose of an overlay is to save storage. An OVERLAY declaration can easily establish a "common" block of working storage for use by different parts of a program for different purposes at different times. For example, if one part of a program uses ALPHA, a floating-point array, while another part uses BETA, a large Hollerith item, while still another part uses GAMMA, a table of intermediate results, and if none of these uses conflicts with any of the other uses, then the following declaration,

```
OVERLAY ALPHA = BETA = GAMMA $
```

by allocating all three data elements to the single block of storage needed for the largest, would save the storage otherwise required for the two remaining.

Another use of overlaying is in the construction of several small items from one large item. To do this, an overlay is declared so that the values of each of the small items are represented by specific portions of the machine symbol representing the value of the large item. The following set of declarations, for example, allow the left and right components of a dual item to be individually designated.

```
ITEM  VECTOR Dual  24 Signed 1Ø Rounded $
ITEM X'COORD fixed 24 Signed 1Ø Rounded $
ITEM Y'COORD fixed 24 Signed 1Ø Rounded $
OVERLAY      VECTOR = X'COORD,Y'COORD $
```

This kind of precise overlaying requires exact knowledge of the amount of storage allocated each of the data elements involved; knowledge that, unfortunately, is not always easily available.

INITIAL VALUE DECLARATIONS. It is often necessary to declare items with specific initial values, in other words, items that, although they may later be assigned other values, must designate particular values when the program initially refers to them, values that are known prior to program compilation. Such items are useful as: parameters that are changed from run to run; as arrays and tables of constants; or as initial data.

The initial value of a simple item may be denoted within the item declaration by a single constant, which must denote a value assignable to the item. This constant, preceded by the Preset descriptor, is usually inserted after the item description but may replace it entirely for numeric and literal values, since item descriptions are somewhat redundant in these two cases.

declaration ≠ ITEM name$_{\text{of-simple-item}}$ ⌊description Preset⌋ constant $

As an example of a preset, parameter item, consider the declaration

```
ITEM ERROR 1.234E-5 $
```

which declares ERROR to be a floating-point item (since the constant is floating-point) with an initial value of 1.234E-5. This item, ERROR, might designate the maximum tolerable error in an arithmetic computation and, though used in many different places, can easily be changed at any recompilation of the program by replacing its declaration with a different one, for example:

```
ITEM ERROR Floating Preset 1.23E-4 $
```

which illustrates the alternative form. Some further examples of parameter item declarations are:

```
ITEM DELTA fixed 17 Signed 3Ø Rounded Preset Ø.5E-5A3Ø $

ITEM DISPLAY'CENTER D(-59.5A5,+Ø5.7A5) $

ITEM KEY'WORD 6H(ABACUS) $

ITEM STATE Status V(FIRST) V(SECOND) V(THIRD) V(FOURTH) V(FIFTH) V(SIXTH)
V(SEVENTH) V(LAST) Preset V(FIRST) $

ITEM OPERATIONAL Boolean Preset Ø $
```

Initial values for a subscripted item may be indicated by appending an array of constants to the array item or table item declaration. Such a constant array, composed of a list either of constants or of constant arrays enclosed in BEGIN and END brackets, denotes a set of values that a subscripted item is to initially designate.

```
constant-array ≠ BEGIN constants;constant-arrays END
```

A one-dimensional constant array consists of a list of constants enclosed in BEGIN and END brackets, for example:

```
BEGIN 8T( UNKNOWN) 8T( PENDING) 8T( HOSTILE) 8T(FRIENDLY) END
```

A two-dimensional constant array consists of a list of one-dimensional constant arrays enclosed in BEGIN and END brackets, for example:

```
BEGIN BEGIN Ø1. Ø2. Ø3. Ø4. Ø5. END
      BEGIN Ø2. Ø4. Ø6. Ø8. 1Ø. END
      BEGIN Ø3. Ø6. Ø9. 12. 15. END
      BEGIN Ø4. Ø8. 12. 16. 2Ø. END
      BEGIN Ø5. 1Ø. 15. 2Ø. 25. END END
```

A three-dimensional constant array consists of a list of two-dimensional constant arrays enclosed in BEGIN and END brackets, and so on. The dimension of the constant array should agree with that of the item it initializes. Thus, both a linear array item and a table item may be initialized by a one-dimensional constant array. For example:

```
ARRAY ALPHA 4 Floating $ BEGIN 1.1498196 .6774323 .2Ø8ØØ3Ø. .1268089 END
```

declares a list of four floating-point coefficients for a third order polynomial, and

```
TABLE       Rigid 12 $
            BEGIN
ITEM  MONTH Hollerith 3 $ BEGIN 3H(JAN) 3H(FEB) 3H(MAR) 3H(APR) 3H(MAY)
            3H(JUN) 3H(JUL) 3H(AUG) 3H(SEP) 3H(OCT) 3H(NOV) 3H(DEC) END
ITEM LENGTH fixed 5 Unsigned $ BEGIN 31 28 31 3Ø 31 3Ø 31 31 3Ø 31 3Ø
            31 END
            END
```

declares a table, which (indexed by month-number - 1) associates a three-letter abbreviation and a number of days for each month in the year. Since a one-dimensional constant array may contain fewer constants than there are items it could initialize, it is important to note that a list of k constants in a constant array will initialize the first k of these items. For instance:

```
ARRAY EMPTY 1ØØØ Boolean $ BEGIN 1 Ø END
```

presets EMPTY($Ø$) to 1, EMPTY(1) to Ø, and leaves the rest undefined. Such partial initialization is useful in providing a routine with just enough initial data to determine whether it is operating correctly, without having to go to the tedious extreme of providing, say, a thousand constants for a thousand element array.

Constant arrays of two or more dimensions serve, of course, to initialize arrays of two or more dimensions. And here, the indexing gets involved. Individual constants in a multi-dimensional constant array are indexed by column number; one-dimensional constant arrays are indexed by row number; two-dimensional constant arrays are indexed by plane number, and so on. To illustrate, the following two-dimensional array of dual fixed-point items is preset so that each item designates its own index value.

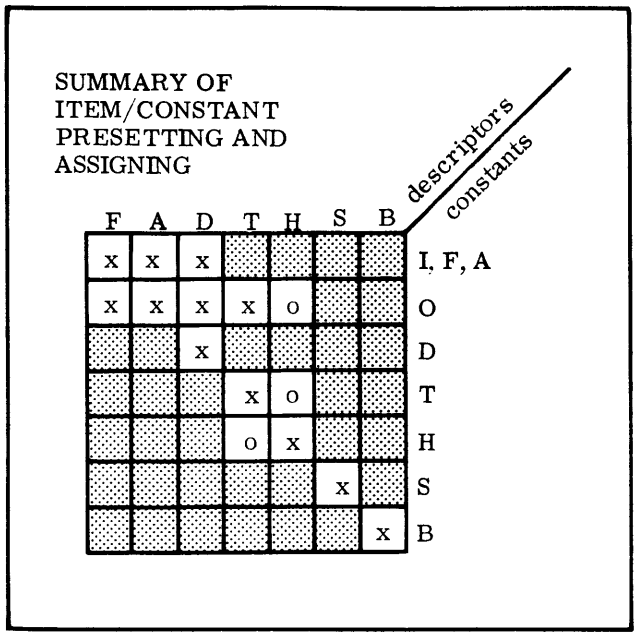
```
ARRAY XY 3 5 Dual 4 Unsigned $
        BEGIN BEGIN  D(Ø,Ø) D(Ø,1) D(Ø,2) D(Ø,3) D(Ø,4)  END
              BEGIN  D(1,Ø) D(1,1) D(1,2) D(1,3) D(1,4)  END
              BEGIN  D(2,Ø) D(2,1) D(2,2) D(2,3) D(2,4)  END END
```

To carry the illustration one step further, the following three dimensional array of literal items is preset so that each item also designates its index value, though in a literal fashion.

```
ARRAY XYZ 2 3 2 Transmission 5 $
          BEGIN BEGIN BEGIN 5H(Ø,Ø,Ø) 5H(Ø,1,Ø) 5H(Ø,2,Ø) END
                      BEGIN 5H(1,Ø,Ø) 5H(1,1,Ø) 5H(1,2,Ø) END END
                BEGIN BEGIN 5H(Ø,Ø,1) 5H(Ø,1,1) 5H(Ø,2,1) END
                      BEGIN 5H(1,Ø,1) 5H(1,1,1) 5H(1,2,1) END END END
```

The constants in a constant array must, of course, denote values that are assignable to the item being initialized, and in addition, though integers and floating and fixed constants may be intermixed, these constants must otherwise all be of the same type. The accompanying chart summarizes the acceptable pairing of item types and constant types -- both for initializing and for assigning. The following abbreviations are used:

I - integer
F - Floating
A - fixed
O - Octal
D - Dual
T - Transmission
H - Hollerith
S - Status
B - Boolean
x - acceptable
o - acceptable but undefined



DEFINE DECLARATIONS. A define declaration, composed of the DEFINE declarator followed by a name and a string of signs enclosed in the '' brackets and terminated by the $ separator, establishes an equivalence between the name and the string of signs by effectively causing the sign string to be substituted for the name wherever it may subsequently occur as a JOVIAL symbol.

declaration ≠ DEFINE name$_{\text{of-signs}}$ '':signs$_{\text{except-the-''-symbol}}$:'' $

The define declaration allows the programmer to make simple additions to the language, to abbreviate lengthy expressions, and to create symbolic parameters. As an example of making simple additions to the language, consider the following set of definitions, which have been tacitly assumed in previous examples,

```
DEFINE  BINARY          ''B'' $     DEFINE  PARALLEL        ''P'' $
DEFINE  BOOLEAN         ''B'' $     DEFINE  PRESET          ''P'' $
DEFINE  DENSE           ''D'' $     DEFINE  RIGID           ''R'' $
DEFINE  DUAL            ''D'' $     DEFINE  ROUNDED         ''R'' $
DEFINE  FIXED           ''A'' $     DEFINE  SERIAL          ''S'' $
DEFINE  FLOATING        ''F'' $     DEFINE  SIGNED          ''S'' $
DEFINE  HOLLERITH       ''H'' $     DEFINE  STATUS          ''S'' $
DEFINE  LIKE            ''L'' $     DEFINE  TRANSMISSION    ''T'' $
DEFINE  MEDIUM          ''M'' $     DEFINE  UNSIGNED        ''U'' $
DEFINE  NO              ''N'' $     DEFINE  VARIABLE        ''V'' $
```

along with the following definitions, which will be tacitly assumed in subsequent examples.

```
DEFINE  IFEITHER        ''IFEITH'' $
DEFINE  POSITION        ''POS'' $
DEFINE  PROCEDURE       ''PROC'' $
```

By means of define declarations, still further additions can be made to JOVIAL to markedly improve its readability. For example:

```
DEFINE  TRUE            ''1''  $     DEFINE  STEP            '',''  $
DEFINE  FALSE           ''Ø''  $     DEFINE  UNTIL           '',''  $
DEFINE  PLUS            ''+''  $     DEFINE  THRU          '',1,''  $
DEFINE  MINUS           ''-''  $     DEFINE  THEN            ''$''  $
DEFINE  MULTIPLIED      ''*''  $     DEFINE  ALSO            ''$''  $
DEFINE  DIVIDED         ''/''  $     DEFINE  TO              ''  ''  $
DEFINE  EXPONENTIATED   ''**'' $     DEFINE  IS              ''  ''  $
DEFINE  EQUAL           ''EQ'' $     DEFINE  BY              ''  ''  $
DEFINE  UNEQUAL         ''NQ'' $     DEFINE  OF              ''  ''  $
DEFINE  GREATER         ''GR'' $     DEFINE  THE             ''  ''  $
DEFINE  LESS            ''LS'' $     DEFINE  WITH            ''  ''  $
DEFINE  NOT'GREATER     ''LQ'' $     DEFINE  THAN            ''  ''  $
DEFINE  NOT'LESS        ''GQ'' $     DEFINE  FROM            ''  ''  $
DEFINE  REPLACED        ''='' $
DEFINE  EXCHANGED       ''=='' $
```

With the above declarations, the following becomes a meaningful JOVIAL statement

```
          BEGIN
          GROSS'PAY OF ($EMPLOYEE$) IS REPLACED WITH HOURS'WORKED OF
($EMPLOYEE$) MULTIPLIED BY PAY'RATE OF ($EMPLOYEE$) $
          IF HOURS'WORKED OF ($EMPLOYEE$) IS GREATER THAN 4Ø THEN
GROSS'PAY OF ($EMPLOYEE$) IS REPLACED WITH GROSS'PAY OF ($EMPLOYEE$)
PLUS (HOURS'WORKED OF ($EMPLOYEE$) MINUS 4Ø) MULTIPLIED BY PAY'RATE
OF ($EMPLOYEE$) DIVIDED BY 2 $
          END
```

The use of the define declaration to abbreviate rather than expand expressions is, of course, also possible. For example, if a particularly lengthy status encoding is used in several item descriptions, the labor of copying it out for each declaration can become unbearable. The solution, obviously, is a define declaration.

```
DEFINE STATE'CODE ''V(ALABAMA) V(ALASKA) V(ARIZONA) V(ARKANSAS) V(CALIFORNIA)
                  V(COLORADO) V(CONNECTICUT) V(DELAWARE) V(FLORIDA) V(GEORGIA)
                  V(HAWAII) V(IDAHO) V(ILLINOIS) V(INDIANA) V(IOWA) V(KANSAS)
                  V(KENTUCKY) V(LOUISIANA) V(MAINE) V(MARYLAND) V(MASSACHUSETTS)
                  V(MICHIGAN) V(MINNESOTA) V(MISSISSIPPI) V(MISSOURI) V(MONTANA)
                  V(NEBRASKA) V(NEVADA) V(NEW'HAMPSHIRE) V(NEW'JERSEY)
                  V(NEW'MEXICO) V(NEW'YORK) V(NORTH'CAROLINA) V(NORTH'DAKOTA)
                  V(OHIO) V(OKLAHOMA) V(OREGON) V(PENNSYLVANIA) V(RHODE'ISLAND)
                  V(SOUTH'CAROLINA) V(SOUTH'DAKOTA) V(TENNESSEE) V(TEXAS)
                  V(UTAH) V(VERMONT) V(VIRGINIA) V(WASHINGTON) V(WEST'VIRGINIA)
                  V(WISCONSIN) V(WYOMING)'' $
```

The savings, even with just two item declarations

```
ITEM BIRTH'PLACE Status STATE'CODE $
ITEM   RESIDENCE Status STATE'CODE $
```

using the STATE'CODE abbreviation, are considerable. The ability to define abbreviations, then, so long as they do not adversely affect readability, can be quite a convenience.

Although the define declaration can be almost spectacularly useful in defining additions and abbreviations, probably its most significant use is in defining symbolic parameters. For example, if a program is written to invert a floating-point matrix of order 5Ø, then a name (say, ORDER) can be defined as equivalent to 5Ø and used in the program wherever a constant denoting the order of the matrix is required.

```
            BEGIN  INVERT'MATRIX
DEFINE ORDER ''5Ø'' $
ARRAY  ALPHA ORDER ORDER Floating Rounded $
      STEP1. FOR I = Ø,1,ORDER-1 $
                   BEGIN
                   .
                   .
                   .
```

Whenever the program must be altered to invert a matrix of a different order, the change becomes trivial: merely the substitution of one definition of ORDER for another.

```
DEFINE ORDER ''6Ø'' $
```

In using define declarations, several points should be remembered: (1) The sign string being defined should contain at least one sign (which may be a blank) but may not contain a '' symbol since this, of course, terminates it; (2) No comments may appear among the symbols of a define declaration without the chance of hopelessly confusing comment and defined sign string, since both are delimited by '' brackets; (3) A defined name should be used only in a context where the sign string it defines will comprise an acceptable JOVIAL form; (4) The substitution of defined sign strings for names occurs during compilation before any other syntactic analysis is made, so that it is possible incautiously to define away needed JOVIAL delimiters; (5) Defined names may themselves appear in sign strings, defined either previously or subsequently, and although this can be useful, the possibility of circular definition is present; (6) A defined name may be redefined at a later point in the program listing, and the latest definition will thereafter be substituted for occurrences of the name.

SPECIFIED-ENTRY-STRUCTURE TABLE DECLARATIONS. It is occasionally necessary to declare a table with a specific and predefined entry structure, as when an input or output message must be declared as a table entry and its format is fixed and part of the specifications of the message processing program. In the table declarations discussed so far, the packing of table items has been left to the JOVIAL compiler. And while the programmer can exercise considerable control over this packing by the No, Medium, or Dense packing descriptors, such control is far from complete. Complete control over the structure of table entries is provided the programmer by the specified-entry-structure table declaration. Such a declaration is very similar to a regular table declaration, with the following exceptions: (1) the table declaration includes number of words per entry; (2) the component item declarations include the index number, within the entry, of the first word containing the item, and the index number, within the word, of the item's origin bit (BIT($Ø$)); the optional No, Medium, or Dense packing descriptor is omitted from the table declaration but may be included in any of the component item declarations to provide the compiler with information on the type of item packing specified for that item; (4) overlay declarations may not be included within the BEGIN and END brackets since the table's items have already been allocated memory space.

declaration ♦ TABLE ⌊name$_{\text{of-table}}$⌋ Variable;Rigid$_{\text{length}}$ number$_{\text{of-entries}}$

⌊Serial;Parallel$_{\text{entry-structure}}$⌋ number$_{\text{of-words-per-entry}}$ $ BEGIN

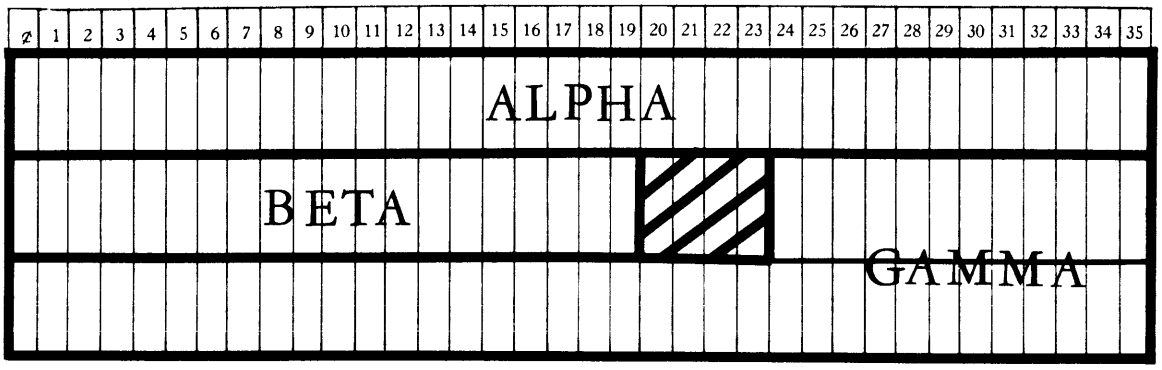
⌈ITEM name$_{\text{of-item}}$ description number$_{\text{index-of-word}}$ number$_{\text{index-of-origin-bit}}$ ⌊No;Medium;Dense$_{\text{item-packing}}$⌋ $ ⌊constant-array⌋⌉s END

The items of a specified-entry-structure table may be initialized in the normal manner with a one-dimensional array of constants.

A specified-entry-structure table declaration is given below, for a computer with a 36-bit word containing six 6-bit sub-words.

```
TABLE          Variable 5ØØ Serial                3                  $
               BEGIN                        ''Word Bit Packing''
ITEM       ALPHA Floating Rounded             Ø    ØØ  No        $
ITEM       BETA fixed 2Ø Signed 5 Ø...1.E4A5   1    ØØ  Dense     $
ITEM       GAMMA Transmission 8               1    25  Medium    $
               END
```

The following entry diagram shows the memory allocation resulting from this declaration



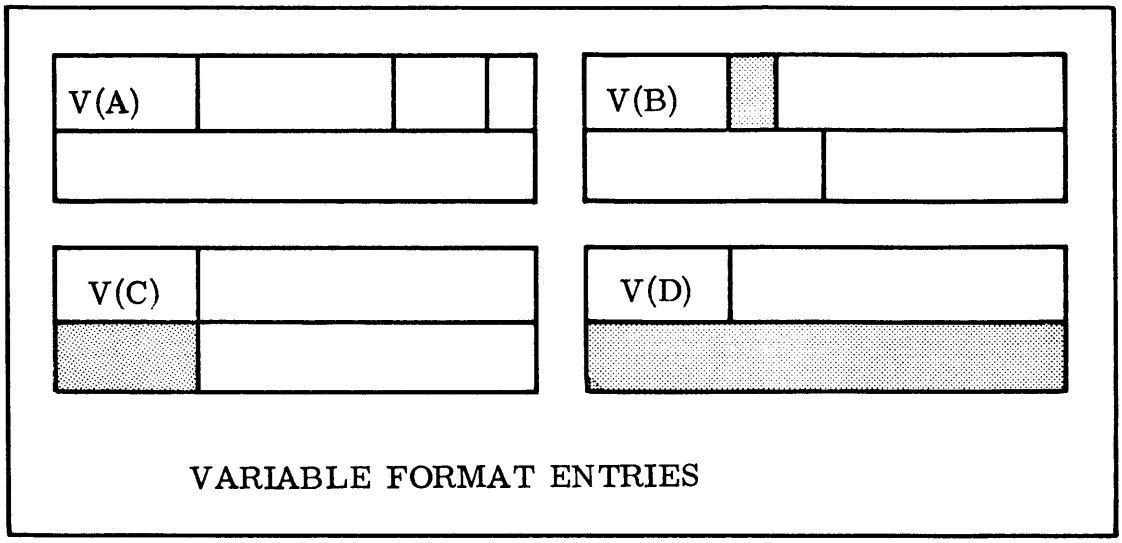
Notice that a floating-point machine symbol takes up a full computer word, and while a Boolean machine symbol requires at least one bit, the sizes of the symbols representing other types of values are given in their descriptions.

Another example of a specified-entry-structure table is given below, in a declaration describing the internal memory of the CDC 16Ø4 computer as a JOVIAL table. Like those of all stored-program digital computers, the 16Ø4's memory words are either instruction words (each containing two single-address instructions -- an upper instruction and a lower instruction) or data words (each containing a single operand -- floating-point, integral, fractional, or logical in type).

```
DEFINE    OPERATION'CODE ''V(HALTØØ) V(ARS) V(QRS)
               V(LRS) V(ENQ) V(ALS) V(QLS) V(LLS) V(ENA) V(INA)
               V(LDA) V(LAC) V(ADD) V(SUB) V(LQC) V(STA)
               V(STQ) V(AJP) V(QJP) V(MUI) V(DVI) V(MUF) V(DVF)
               V(FAD) V(FSB) V(FMU) V(FDV) V(SCA) V(SCQ) V(SSK)
               V(SSH) V(SST) V(SCL) V(SCM) V(SSU) V(LDL) V(ADL)
               V(SBL) V(STL) V(ENI) V(INI) V(LUI) V(LIL) V(ISK)
               V(IJP) V(SIU) V(SIL) V(SAU) V(SAL) V(INT) V(OUT)
               V(EQS) V(THS) V(MEQ) V(MTH) V(RAD) V(RSB) V(RAO)
               V(RSL) V(EXF) V(SLJ) V(SLS) V(HALT77)'' $

TABLE     CDC'16Ø4'MEMORY Rigid                          1        $
               BEGIN                                  ''Word Bit''
               ''Instruction Word''
ITEM      UPPER'OPERATION Status 6 OPERATION'CODE        Ø    ØØ  $
ITEM      UPPER'INDEX fixed 3 Unsigned                   Ø    Ø6  $
ITEM      UPPER'CONDITION Status 3 V(JØ) V(J1) V(J2)
               V(J3) V(RØ) V(R1) V(R2) V(R3)             Ø    Ø6  $
ITEM      UPPER'ADDRESS fixed 15 Unsigned                Ø    Ø9  $
ITEM      UPPER'COUNT fixed 15 Unsigned Ø...127          Ø    Ø9  $
ITEM      UPPER'OPERAND fixed 15 Signed                  Ø    Ø9  $
ITEM      LOWER'OPERATION Status 6 OPERATION'CODE        Ø    24  $
ITEM      LOWER'INDEX fixed 3 Unsigned                   Ø    3Ø  $
ITEM      LOWER'CONDITION Status 3 V(JØ) V(J1) V(J2)
               V(J3) V(RØ) V(R1) V(R2) V(R3)             Ø    3Ø  $
ITEM      LOWER'ADDRESS fixed 15 Unsigned                Ø    33  $
ITEM      LOWER'COUNT fixed 15 Unsigned Ø...127          Ø    33  $
ITEM      LOWER'OPERAND fixed 15 Signed                  Ø    33  $
               ''Data Word''
ITEM      FLOATING'OPERAND Floating Rounded              Ø    ØØ  $
ITEM      INTEGRAL'OPERAND fixed 48 Signed               Ø    ØØ  $
ITEM      FRACTIONAL'OPERAND fixed 48 Signed 47          Ø    ØØ  $
ITEM      LOGICAL'OPERAND fixed 48 Unsigned              Ø    ØØ  $
               END
```

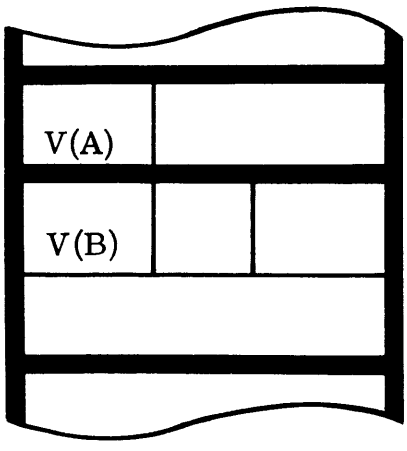
VARIABLE ENTRIES. With the exception of the table in the last example, most of the tables appearing in this manual have been "homogeneous" tables -- homogeneous in the sense that each entry in such a table has the same structure. The CDC 16Ø4 MEMORY table was an inhomogeneous table, however, since some of its entries had instruction word formats while others had data word formats, and even within these two categories there were more than one format. For obvious reasons, the entries in tables such as this are termed "variable format" entries. (Notice, in the diagram below, the common item whose values distinguish between the various structures.)



VARIABLE FORMAT ENTRIES

Using overlay declarations, it is, of course, also possible to overlay different, compiler-created entry formats. Tables of data on dissimilar "objects" can thus be declared in either way. However, the entries in such tables, though variable in format, are all the same length. And in tables where different entries can contain widely differing amounts of data, fixed-length entries are quite wasteful of storage space.

Tables with variable entry formats that also vary in length can be declared as specified-entry-structure, one-word-per-entry tables. Such tables are indexed by word rather than by entry to allow entries to begin with any word in the table. In using a FOR loop to cycle through a table with variable length entries, the subscript must be incremented by the number of words in the current entry in order to obtain the index of the next entry. In the accompanying diagram, for example, if the current repetition of a loop was processing a one-word entry (whose common, control item specifies the status value A), then the correct index increment would be one; but if the current repetition was processing a two-word entry (distinguished by the status value B), then the correct increment would be two.



VARIABLE LENGTH ENTRIES

As an example of a table with variable-length entries, consider the following COMPOOL table for data elements -- tables, items, arrays, and strings (to be discussed in the next section). Each entry in this table completely describes one data element. Just three items, LABEL, TYPE, and WORDS'THIS'ENTRY are common to every entry in the table, while the other items only appear in an entry if they are appropriate for the type of data element described. Thus, the structure and length of an entry in this table depends on the data element it describes.

```
TABLE       DATA'POOL Variable 3ØØØ                          1                  $
                BEGIN                                  ''Word Bit Packing''
ITEM        LABEL Hollerith 6                            Ø    ØØ  No          $
ITEM        CODING Status 3 V(FIXED) V(FLOATING)
                V(TRANSMISSION) V(HOLLERITH)
                V(STATUS) V(BOOLEAN)                     1    ØØ  Medium      $
ITEM        ADDRESS fixed 15 Unsigned                    1    Ø3  Medium      $
ITEM        TYPE Status 3 V(TABLE) V(ITEM) V(ARRAY)
                V(STRING)                                1    18  Medium      $
ITEM        BEADS'PER'WORD fixed 6 Unsigned              1    21  Dense       $
ITEM        WORDS'THIS'ENTRY fixed 9 Unsigned            1    27  Dense       $
ITEM        WORD'INDEX fixed 15 Unsigned                 1    Ø3  Medium      $
ITEM        SIGNED Boolean                               2    ØØ  Dense       $
ITEM        PACKING Status 2 V(NO) V(MEDIUM)
                V(DENSE)                                 2    Ø1  Dense       $
ITEM        VARIABLE'LENGTH Boolean                      2    Ø2  Medium      $
ITEM        NUMBER'OF'ENTRIES fixed 15 Unsigned          2    Ø3  Medium      $
ITEM        STRING'INTERVAL fixed 6 Unsigned             2    Ø3  Dense       $
ITEM        NUMBER'OF'DIMENSIONS fixed 9 Unsigned        2    Ø9  Dense       $
ITEM        NUMBER'OF'STATUSES fixed 9 Unsigned          2    Ø9  Dense       $
ITEM        FRACTION'BITS fixed 6 Unsigned               2    18  Dense       $
ITEM        PARALLEL'ENTRY Boolean                       2    2Ø  Medium      $
ITEM        WORDS'PER'ENTRY fixed 15 Unsigned            2    21  Medium      $
ITEM        BITS'PER'ITEM fixed 6 Unsigned               2    24  Dense       $
ITEM        ORIGIN'BIT fixed 6 Unsigned                  2    3Ø  Dense       $
ITEM        PRESET'VALUE fixed 36 Unsigned               3    ØØ  No          $
ITEM        TABLE'NAME Hollerith 6                       3    ØØ  No          $
ITEM        DIMENSION'LIST'INDEX fixed 12 Unsigned       4    ØØ  Dense       $
ITEM        STATUS'LIST'INDEX fixed 12 Unsigned          4    12  Dense       $
ITEM        CONSTANT'LIST'INDEX fixed 12 Unsigned        4    24  Dense       $
                END
```

The DATA'POOL table above is based on the memory structure of the IBM 7090 computer, which has a 36-bit word divided 3/15/3/15 into four subwords. A loop for cycling through this table could be created with the following FOR statement.

```
FOR I = Ø,WORDS'THIS'ENTRY($I$),2999 $
```

STRING ITEM DECLARATIONS. A string is an item occurring in a specified-entry-structure table not just once per entry, but many times per entry. The number of occurrences (or "beads") of a string item can be allowed to vary from entry to entry, thus creating a table with variable length entries, but in this case, a control item must be declared in which a count is kept of the number of beads in each entry.

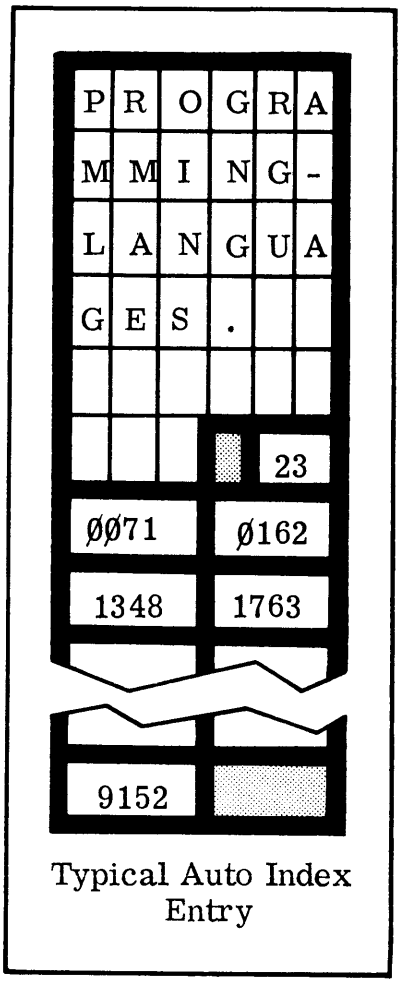
A string item declaration is very similar in form to the declaration of other specified-entry-structure table items. Instead of the declarator ITEM, however, the declarator STRING is used, and two additional factors are appended to the declaration: (1) an interval factor giving the number of words from the first word of the entry containing beads of the string to the next such word; (2) a packing factor giving the number of beads per word.

declaration ≠ STRING name$_{\text{of-string-item}}$ description number$_{\text{index-of-word}}$ number$_{\text{index-of-origin-bit}}$ ⌊No;Medium;Dense$_{\text{item-packing}}$⌋ number$_{\text{of-intervening-words}}$ number$_{\text{of-beads-per-word}}$ $ ⌊constant-array⌋

To illustrate the string item concept, consider the table AUTO'INDEX, each item of which contains a topic phrase and a list of reference numbers of pertinent documents.

```
TABLE  AUTO'INDEX Variable 2ØØØØ   1                              $
           BEGIN                  ''Word Bit Pack  Skip B/W''
ITEM   TOPIC Hollerith 33          Ø    ØØ  Dense                $
ITEM   COUNT fixed 12 Unsigned     4    24  Dense                $
STRING REFERENCE fixed 18 Unsigned 5    ØØ  Dense 1    2       $
           END
```

A typical entry in the AUTO'INDEX table might contain the topic phrase "PROGRAMMING-LANGUAGES." and 23 document references, ranging from document #ØØ71 to document #9152. The string item, REFERENCE, is packed in consecutive words, two beads per word.



Typical Auto Index Entry

In designating the value of any particular bead, a two-component index must subscript the string item's name specifying, first, the bead and, second, the entry. Thus, if the entry in the accompanying diagram is entry I, beginning at word I in the table (entry I and word I are synonymous here, since the table is a one-word-per-entry table) then

REFERENCE($Ø,I$)

specifies the value 71, and

REFERENCE($22,I$)

specifies 9152. Notice that the AUTO'INDEX table's actual number of words per entry is 6+(COUNT(I)+1)/2, discarding any fractional remainder from the division. Thus

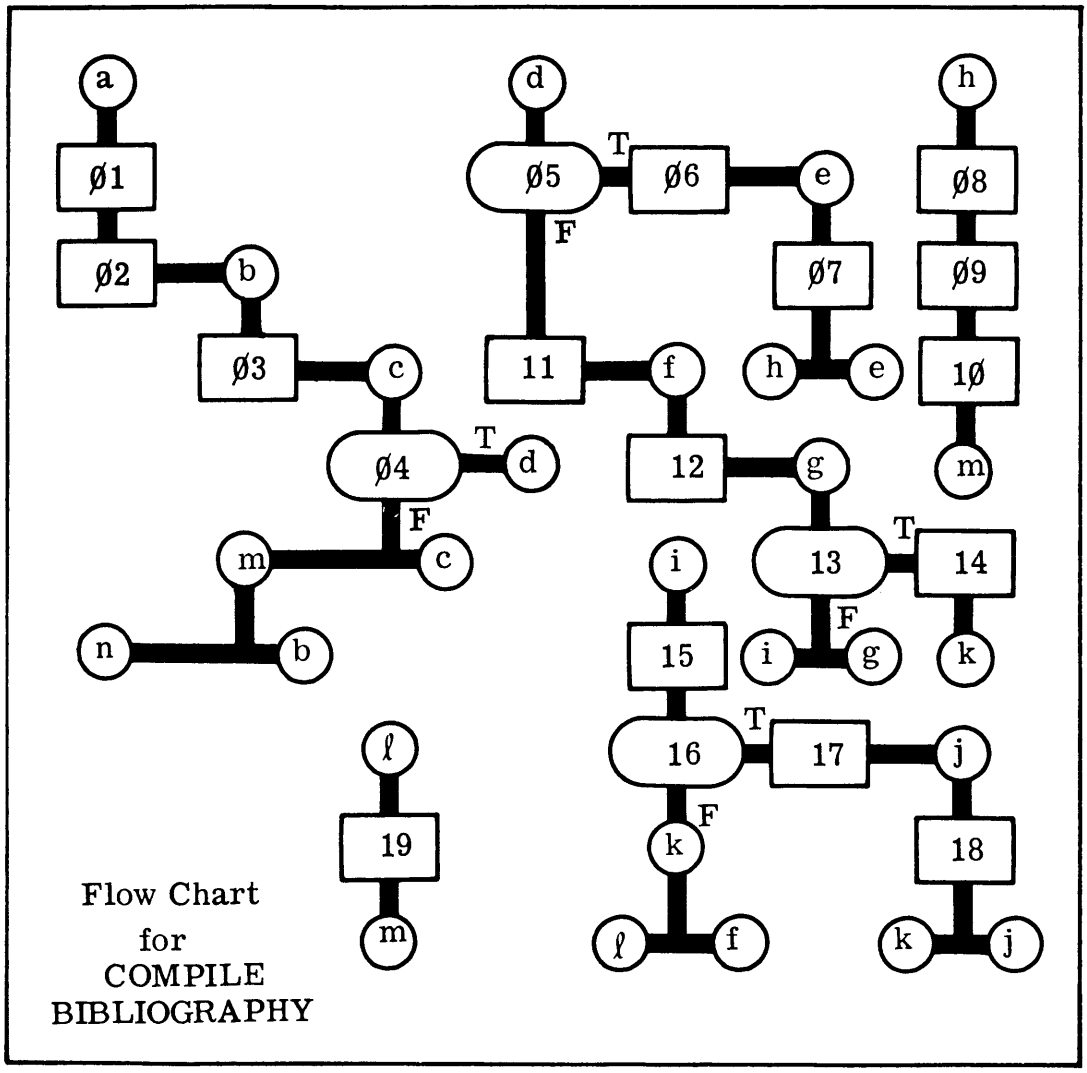
FOR I = Ø,6+(COUNT(I)+1)/2,19999 $

would create a FOR loop to cycle through the table and process each entry.

To show how string items are manipulated, the following routine searches the AUTO'INDEX table and compiles a bibliography list of reference numbers of those documents pertaining to all the topics in a given list of topics. (For example, if the given list of topics contained just "PROGRAMMING-LANGUAGES." and "PROCEDURE-ORIENTED.", the bibliography list would contain just those references pertaining to both topics.)

```
          BEGIN COMPILE'BIBLIOGRAPHY.
ARRAY     SEARCH'TOPIC 256 Hollerith 33 $
ITEM      NUMBER'OF'SEARCH'TOPICS fixed 8 Unsigned $
ARRAY     PERTINENT'REFERENCE 4Ø96 fixed 18 Unsigned $
ITEM      NUMBER'OF'PERTINENT'REFERENCES fixed 8 Unsigned $
ITEM      FIRST'SEARCH'TOPIC Boolean $
STEPØ1.   FIRST'SEARCH'TOPIC = 1 $
STEPØ2.   FOR H = Ø,1,NUMBER'OF'SEARCH'TOPICS-1 $
               BEGIN
STEPØ3.        FOR I = Ø,6+(COUNT($I$)+1)/2,19999 $
                    BEGIN
STEPØ4.             IF SEARCH'TOPIC($H$) EQ TOPIC($I$) $
                         BEGIN
STEPØ5.                  IF FIRST'SEARCH'TOPIC $
                              BEGIN
STEPØ6.                       FOR J = Ø,1,COUNT($I$)-1 $
STEPØ7.                            PERTINENT'REFERENCE($J$) =
                                   REFERENCE($J,I$) $
STEPØ8.                       FIRST'SEARCH'TOPIC = Ø $
STEPØ9.                       NUMBER'OF'PERTINENT'REFERENCES =
                              COUNT($I$) $
STEP1Ø.                       TEST H $
                              END
STEP11.                  FOR X = Ø,1,NUMBER'OF'PERTINENT'REFERENCES-1 $
                              BEGIN
STEP12.                       FOR Y = Ø,1,COUNT($I$)-1 $
                                   BEGIN
STEP13.                            IF PERTINENT'REFERENCE($X$) EQ
                                   REFERENCE($Y,I$) $
STEP14.                                 TEST X $
                                   END
STEP15.                       NUMBER'OF'PERTINENT'REFERENCES =
                              NUMBER'OF'PERTINENT'REFERENCES - 1 $
STEP16.                       IF X LS NUMBER'OF'PERTINENT'REFERENCES $
                                   BEGIN
STEP17.                            FOR Z = X,1,
                                   NUMBER'OF'PERTINENT'REFERENCES $
STEP18.                                 PERTINENT'REFERENCE($Z$) =
                                        PERTINENT'REFERENCE($Z+1$) $
                              END  END
STEP19.                  TEST H $
          END  END  END  END
```

The following flow chart shows the sequence of statement executions for this routine. Steps 6 thru 9 create a list of possible references from those associated with the first topic found in the AUTO'INDEX table, and steps 11 thru 18 eliminate those references that do not occur with each of the other topics found in the table.



MISCELLANEOUS STATEMENTS

STOP STATEMENTS. A STOP statement, composed of the STOP sequential operator followed by an optional statement name and terminated by the $ separator, halts or indefinitely delays the sequence of statement executions.

statement ≠ STOP [name$_{\text{of-next-statement}}$] $

A STOP statement usually indicates an operational end to the program in which it appears, and no further program control exists until the computer operator reinitiates it. However, if the program is restarted, execution will resume either with the next statement listed, if no statement name is given in the STOP statement, or with the indicated statement, if a statement name is provided.

STOP statements are seldom used in system programs, since such programs usually terminate their operation by transferring the sequence of statement executions to the system control program. The control program itself, however, may use the STOP statement, for example: to give the computer operator time to act after a programmed request for operator intervention, such as "READY MORE INPUT AND RESUME COMPUTATION." logged out on the on-line printer.

Two examples of STOP statements are given below. Notice that, in conjunction with the GOTO statement, the effect of the first is identical to that of the second.

```
STOP $  GOTO STEP1 $

STOP STEP1 $
```

DIRECT-CODE STATEMENTS. A direct-code statement allows the programmer to include a routine coded in a "direct" or machine-oriented programming language among the statements of a JOVIAL program. This can occasionally be quite a convenience, especially if the programmer wants to take advantage of some of the more exotic features of the computer on which his program will run. A direct-code statement is composed of a sequence of machine-oriented, symbolic instructions enclosed in the DIRECT and JOVIAL brackets. Such a sequence of instructions is an arbitrary string of signs as far as JOVIAL is concerned, and the statement itself has a computer dependent and therefore undefined operational effect.

statement ¢ DIRECT signs$_{\text{except-the-JOVIAL-bracket}}$ JOVIAL

Machine-oriented programming languages are somewhat outside the scope of this manual, so no examples of direct-code statements will be given here. However, it is possible to designate the values of JOVIAL items within a directly coded routine -- by using the JOVIAL-like ASSIGN statement.

An ASSIGN statement has a format similar to that of a regular JOVIAL assignment statement, but preceded by the ASSIGN symbol. The other two elements of this statement, aside from the = and $ separators, are interchangeable and consist of a JOVIAL variable designating an item value, and a non-JOVIAL variable designating the value represented by the contents of the computer's accumulator -- an undefined machine register. The syntax of the ASSIGN statement is indicated below.

ASSIGN A(:⌊integer⌋:) = name$_{\text{of-item}}$ ⌊($ index $)⌋ $

ASSIGN name$_{\text{of-item}}$ ⌊($ index $)⌋ = A(:⌊integer⌋:) $

The letter A, followed by an optional integer enclosed in the (and) parentheses, which may not be omitted, designates the value in the accumulator. The precision of this value is given in number of fractional bits by the integer, which is usually zero for all non-numeric and integral values, and omitted entirely for floating-point values. The other half of the ASSIGN statement can designate the value of a subscripted as well as a simple item, so it should be noted that there are no limitations on the complexity of the numeric formulas in an index subscripting the item name. All in all, the rules of the ASSIGN statement are the same as those of the JOVIAL assignment statement, even to the automatic conversion between fixed and floating-point representation.

Some examples of the ASSIGN statement are given below:

```
ASSIGN A() = ALPHA($I,J**2$) $
```

assigns the specified numeric value of ALPHA to the accumulator as a floating-point machine symbol.

```
ASSIGN A(1Ø) = BETA $
```

assigns the numeric value of BETA to the accumulator as a fixed-point machine symbol with 1Ø fractional bits.

```
ASSIGN GAMMA = A(-6) $
```

assigns the value represented in the accumulator by a fixed-point machine symbol, whose least significant bit is precise to 64 units, to be the value designated by GAMMA.

ASSIGN WORD($Ø$) = A(Ø) $

assigns the literal value in the accumulator to be the value designated by the first WORD.

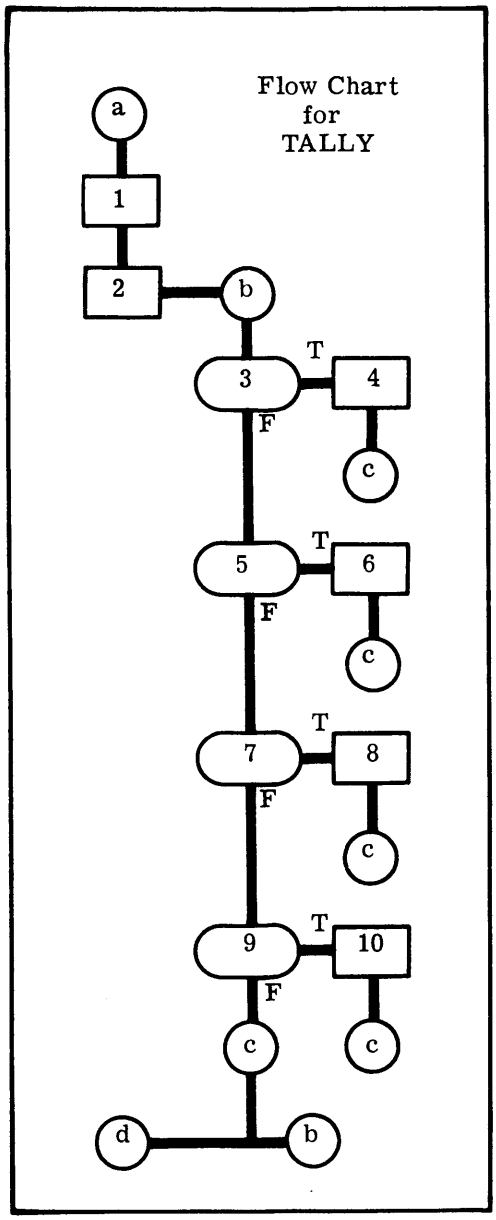
ALTERNATIVE STATEMENTS. An alternative statement, composed of a sequence of IF-like IFEITHer-ORIF substatements each followed by its own conditional statement and finally terminated by the END bracket, selects for execution from its set of conditional statements the one associated with the first True Boolean formula from the corresponding set of IFEITHer-ORIF substatements. The effect of an alternative statement is therefore equivalent to that of the selected statement by itself.

statement ≡ IFEITHer ⌈boolean-formula $ statement⌉$_{s}^{ORIF}$ END

To illustrate the use of the alternative statement, consider the following example, which (given floating-point items P1, P2, and P3, and a linear, floating-point array ALPHA of length N'ALPHA, and assuming P1 LS P2 LS P3) counts values of ALPHA that are (a) less than P1, (b) less than P2 but not less than P1, (c) less than P3 but not less than P2, (d) not less than P3.

```
            BEGIN TALLY.
MODE        fixed 15 Unsigned $
     STEP1. TALLY'A = Ø $  TALLY'B = Ø $  TALLY'C = Ø $  TALLY'D = Ø $
     STEP2. FOR A = Ø,1,N'ALPHA-1 $
     STEP3.      IFEITHer ALPHA($A$) LS P1 $
     STEP4.           TALLY'A = TALLY'A+1 $
     STEP5.      ORIF P1 LQ ALPHA($A$) LS P2 $
     STEP6.           TALLY'B = TALLY'B+1 $
     STEP7.      ORIF P2 LQ ALPHA($A$) LS P3 $
     STEP8.           TALLY'C = TALLY'C+1 $
     STEP9.      ORIF P3 LQ ALPHA($A$) $
    STEP1Ø.           TALLY'D = TALLY'D+1 $
            END  END
```

The accompanying flow chart, showing the execution sequence of the statement TALLY, is typical of alternative statements. Notice, however, that the set of Boolean formulas in steps 3, 5, 7, and 9 exhaust all the possibilities, so that if the formulas in steps 3, 5, and 7 all specify False, then the formula in step 9 must specify True, and the False branch from that statement is never used. This can be made more apparent by replacing the statement with




```
STEP9. ORIF 1 $
```

which is also more efficient, since no comparison is made. And by using a define declaration, this particular construction, which is not uncommon, can be made much more readable.

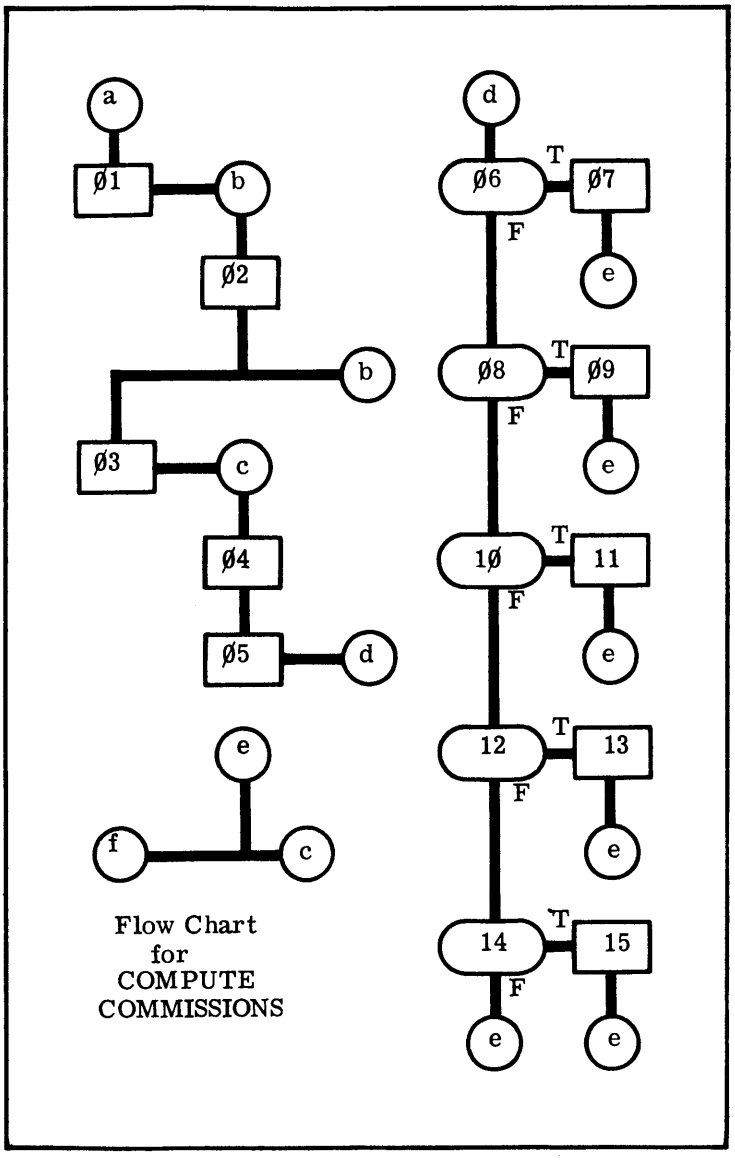
```
DEFINE OTHERWISE ''ORIF 1 $'' $
STEP9. STEP1Ø. OTHERWISE
TALLY'D = TALLY'D+1 $
```

Since the normal successor to each of the conditional statements within an alternative statement is the statement listed after the alternative statement, it is important to realize that no more than one of these conditional statements will be executed, no matter how many of their corresponding Boolean formulas specify True. (Of course, if none of the Boolean formulas specify True, none of the conditional statements will be executed and the alternative statement will have no operational effect.) It is also important to note that, although an ORIF substatement can be named, it is not actually a complete statement and cannot, therefore, be combined with a preceding IF or FOR statement.

As another illustration of the alternative statement, consider the following example, which computes salesman's commissions.

```
                    BEGIN COMPUTE'COMMISSIONS.
TABLE SALES         Variable 1ØØØØ Serial Dense $
                         BEGIN
ITEM  ARTICLE            fixed 1Ø Unsigned $''Entry index to ARTICLES''
ITEM  SALESMAN           fixed Ø7 Unsigned $''Entry index to SALESMEN''
                         END
TABLE ARTICLES      Variable 1ØØØ Serial Dense $
                         BEGIN
ITEM  COST               fixed 36 Unsigned $''In cents''
ITEM  SALE'PRICE         fixed 36 Unsigned $''In cents''
                         END
TABLE SALESMEN      Variable 1ØØ Serial Dense $
                         BEGIN
ITEM  NET'COMMISSIONS fixed 27 Unsigned $''In cents''
ITEM  COMMISSION'PLAN Status V(A) V(B) V(C) V(D) V(E) $
                         END
                    FOR S = ALL (SALESMEN) $
                         NET'COMMISSIONS($S$) = Ø $
                    FOR I = ALL (SALES) $
                         BEGIN
                         FOR A = ARTICLE($I$) $
                         FOR S = SALESMAN($I$) $
                              IFEITHer COMMISSION'PLAN($S$) EQ V(A) $
                                   NET'COMMISSIONS($S$) = NET'COMMISSIONS($S$)
                    + .15A15 * SALE'PRICE($A$) $
                              ORIF COMMISSION'PLAN($S$) EQ V(B) $
                                   NET'COMMISSIONS($S$) = NET'COMMISSIONS($S$)
                    + .4ØA15 * (SALE'PRICE($A$)-COST($A$)) $
                              ORIF COMMISSION'PLAN($S$) EQ V(C) $
                                   NET'COMMISSIONS($S$) = NET'COMMISSIONS($S$)
                    + .1ØA15 * (COST($A$) + (SALE'PRICE($A$)-COST($A$))) $
                              ORIF COMMISSION'PLAN($S$) EQ V(D) $
                                   NET'COMMISSIONS($S$) = NET'COMMISSIONS($S$)
                    + .Ø5A15 * COST($A$) + 1ØØØ $
                              ORIF COMMISSION'PLAN($S$) EQ V(E) $
                                   NET'COMMISSIONS($S$) = NET'COMMISSIONS($S$)
                    + 1500 $  END
                    END  END
```

The flow chart for this statement is shown below.



Flow Chart
for
COMPUTE
COMMISSIONS

CLOSED STATEMENTS. Often, the same list of statements is needed at several different places in a program. One solution would be to define the list (with a define declaration) and use the defined name in the program wherever the list of statements is required. It seems intuitively wasteful, however, to repeat the same statements many times throughout the program and thus generate many identical sequences of machine instructions, so another solution is to write the list of

statements once, and call for its execution wherever in the program it is required. This function is performed with the closed statement.

A closed statement, composed of the CLOSE sequential operator followed by the name of the statement, the $ separator, and the statement itself, is, in effect, removed from the normal, listed sequence of statement executions, and may be correctly invoked only by a GOTO statement. The normal successor to a closed statement is the statement listed after the invoking GOTO statement.

statement ∉ CLOSE name$_{\text{of-statement}}$ $ statement

A closed statement is a closed and parameterless subroutine -- closed in the sense of being removed from the statement execution sequence, and parameterless in the sense that it must already "know" the data on which it is to operate, since its operation may not be adjusted at execution time.

As an example of a closed statement, consider SHELL'SORT below.

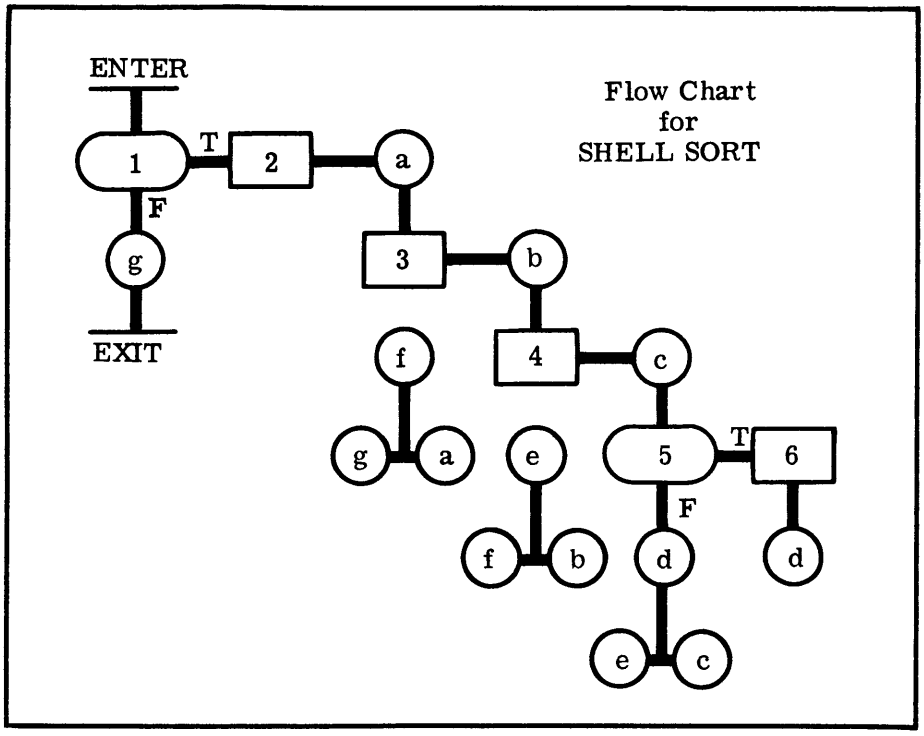
```
CLOSE SHELL'SORT $''A closed statement which sorts a table's entries by
                  KEY item, using Shell's sorting algorithm, as described
                  in ACM Communications - July 59.''
                  BEGIN
DEFINE       KEY  ''          '' $''To be filled in with the name of a
                  table item by the user of this routine before its
                  compilation.''
           STEP1. IF NENT(KEY) GR 1 $
                       BEGIN
           STEP2.      FOR M = NENT(KEY)/2,-(M+1)/2,1 $
                            BEGIN
           STEP3.           FOR J = 1,1,NENT(KEY)-M $
                                 BEGIN
           STEP4.                FOR I = J-1,-M,Ø $
                                       BEGIN
           STEP5.                      IF KEY($I$) GR KEY($I+M$) $
           STEP6.                           ENTRY (KEY($I$)) == ENTRY
(KEY($I+M$)) $    END  END  END  END  END
```

The user of SHELL'SORT would copy it into his program, filling in the define declaration with the name of the table item on which the table is to be ordered. Then, whenever the table must be sorted, the statement

GOTO SHELL'SORT $

will cause the sorting to be done.

The flow chart below shows the statement execution sequence for SHELL'SORT.

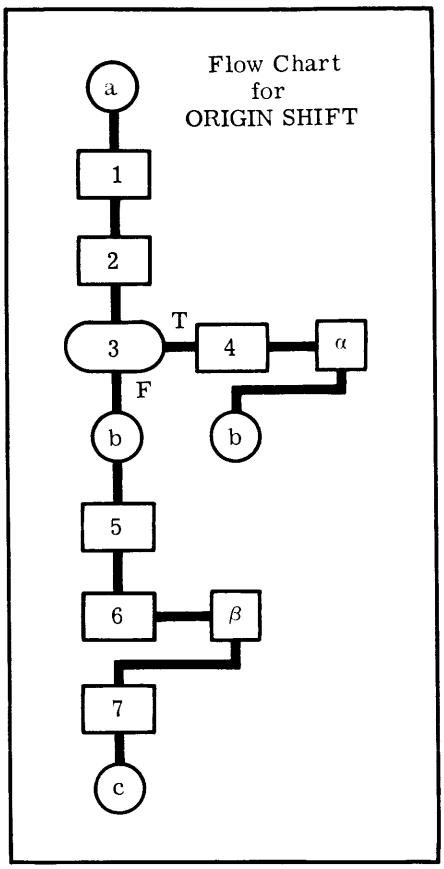


As further examples, consider the following set of closed statements, which convert between longitude and latitude, and rectangular x,y coordinates.

```
                    ''Lon,Lat/x,y Coordinate Conversion Subroutines''
ITEM LONORG'LATORG Dual  16 Signed 7 $''Longitude,Latitude of Origin in
                    degrees.  East,North=plus, West,South=minus.''
ITEM LONORG         fixed 16 Signed 7 $
ITEM        LATORG  fixed 16 Signed 7 $
OVERLAY             LONORG'LATORG = LONORG,LATORG $
ITEM       LON'LAT  Dual  16 Signed 7 $''Longitude,Latitude of point in
                    degrees.''
ITEM       LON      fixed 16 Signed 7 $
ITEM           LAT  fixed 16 Signed 7 $
OVERLAY             LON'LAT = LON,LAT $
ITEM     XPOS'YPOS  Dual  16 Signed 5 $''X,Y coordinates of point in
                    nautical miles from origin.''
ITEM     XPOS       fixed 16 Signed 5 $
ITEM          YPOS  fixed 16 Signed 5 $
OVERLAY             XPOS'YPOS = XPOS,YPOS $
DEFINE         EDE  ''6883.46Ø26A18'' $''Equatorial Diameter of the
                    Earth, in nautical miles.''
DEFINE           PØ ''3.1415927A28/1Ø8ØØ'' $
DEFINE           P1 ''EDE/2*PØ'' $
DEFINE           P2 ''-P1*PØ'' $
DEFINE           P3 ''P1'' $
DEFINE           P4 ''P1/4*PØ'' $
ITEM         K1'K3  Dual  16 Signed 14 $
ITEM         K1     fixed 16 Signed 14 $
ITEM            K3  fixed 16 Signed 14 $
OVERLAY             K1'K3 = K1,K3 $
ITEM         K2'K4  Dual  16 Signed 14 $
ITEM         K2     fixed 16 Signed 14 $
ITEM            K4  fixed 16 Signed 14 $
OVERLAY             K2'K4 = K2,K4 $
ITEM         T1'T2  Dual  16 Signed Ø7 $
ITEM         T1     fixed 16 Signed Ø7 $
ITEM            T2  fixed 16 Signed Ø7 $
OVERLAY             T1'T2 = T1,T2 $
ITEM         BN'AN  Dual  16 Signed Ø7 $
ITEM         BN     fixed 16 Signed Ø7 $
ITEM            AN  fixed 16 Signed Ø7 $
OVERLAY             BN'AN = BN,AN $
```

```
                    CLOSE
LONGITUDE'ADAPTATION $''Adapts the K-parameters for the longitude of
                    any given Origin.''
                    BEGIN
                    K1 = P1*COS(LONORG) $
                    K2 = P2*SIN(LONORG) $
                    K3 = P3 $
                    K4 = P4*SIN(2*LONORG) $
                    END
                    CLOSE
    LONLAT'TO'XY $ ''Converts Longitude,Latitude to x,y''
                    BEGIN
                    T1 = LAT $
                    T2 = LON $
                    XPOS'YPOS = K1'K3*LON'LAT+K2'K4*T1'T2*LON $
                    END
                    CLOSE
    XY'TO'LONLAT $ ''Converts x,y to Longitude,Latitude''
                    BEGIN
                    BN'AN = D(Ø.A7,Ø.A7) $
    XY'TO'LONLAT1. LON'LAT = BN'AN $
                    T1 = AN $
                    T2 = BN $
                    BN'AN = (XPOS'YPOS+T1'T2*BN-K2'K4)/K1'K3 $
                    IF (/BN'AN-LON'LAT/) GQ D(.5A7,.5A7) $
                        GOTO XY'TO'LONLAT1 $
                    LON'LAT = (LONORG'LATORG+(BN'AN+LON'LAT)/D(+2,-2))
/ D(6Ø,6Ø) $        END
```

The above coordinate conversion subroutines presume, of course, the existence of appropriate SIN and COS functions. The solution to the following problem illustrates how these closed statements would be used. Given points a and b whose locations are designated in degrees of longitude and latitude by the dual items LONA'LATA and LONB'LATB, and point c whose location is designated by x,y coordinates in nautical miles from point b by the dual item XBC'YBC. Compute the location of point c in x,y coordinates in nautical miles from point a, to be designated by the dual item XAC'YAC.




```
       BEGIN ORIGIN'SHIFT.
ITEM NEW'LONGITUDE Boolean $
STEP1. NEW'LONGITUDE = LONORG NQ LONA $
STEP2. LONORG'LATORG = LONA'LATA $
STEP3. IF NEW'LONGITUDE $
STEP4.      GOTO LONGITUDE'ADAPTATION $
STEP5. LON'LAT = LONB'LATB $
STEP6. GOTO LONLAT'TO'XY $
STEP7. XAC'YAC = XBC'YBC+XPOS'YPOS $
       END
```

In the accompanying flow chart for ORIGIN'SHIFT, note that α and β refer to other, omitted flow charts -- i.e., those for LONGITUDE'ADAPTATION and LONLAT'TO'XY.

In using any closed statement, it is the programmer's responsibility to see that it is entered only by a GOTO statement referring to it by name, never by a name labeling one of the statements within it, and never as part of the normal listed sequence of statement executions. Furthermore, although a closed statement may call other closed statements, it may not call itself, either directly or indirectly.

RETURN STATEMENTS. A closed statement, and a procedure as well, is enclosed in two automatically generated and inserted routines -- an entrance, and an exit. The entrance routine saves the memory location of the statement invoking the closed statement or procedure, and the exit routine uses this memory location to return to the main sequence of statement executions, immediately after the invoking statement. Since the exit routine is an implied statement, similar in this respect to the implicit Modify-Test-Repeat statement inserted at the bottom of FOR loops, the problem of

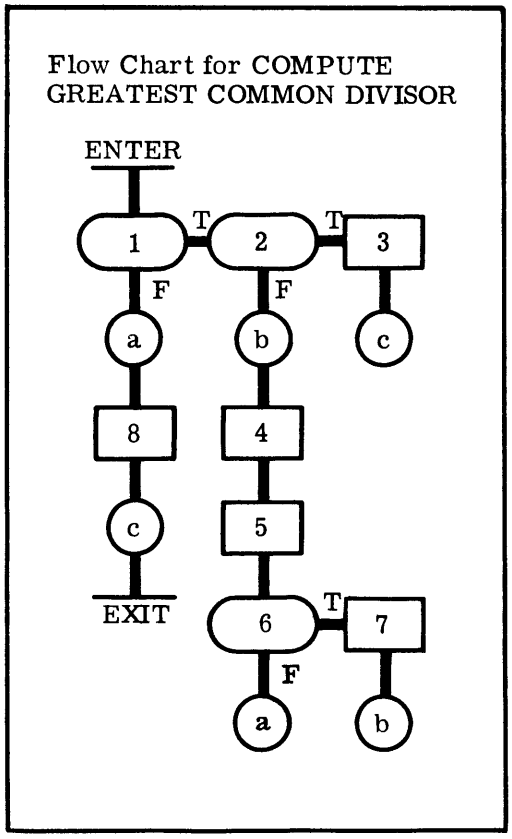
transferring execution control to it from the middle of a closed statement or procedure is similar to that solved in FOR loops by the TEST statement. In closed statements and procedures, however, this function is performed by the RETURN statement.

A RETURN statement, composed of the RETURN sequential operator and terminated by the $ separator, indicates a jump to the exit routine that is automatically inserted after the last listed statement of the closed statement or procedure. A RETURN statement may therefore appear only within a closed statement or a procedure.

statement ≡ RETURN $

Step 3 in the following closed statement shows how RETURN statements are used.

```
       CLOSE
COMPUTE'GREATEST'COMMON'DIVISOR $
''Every pair of numbers ALPHA and
BETA not both zero have a positive
greatest common divisor GAMMA.
This closed statement uses the
Euclidean algorithm as given by
Claussen in the April 60 ACM
Communications.''
       BEGIN
MODE Floating Rounded $
OVERLAY GAMMA = TEMP $
STEP1. IF ALPHA NQ 0. $
            BEGIN
STEP2.      IF BETA EQ 0. $
STEP3.           BEGIN
                 GAMMA = ALPHA $
                 RETURN $
                 END
STEP4.      FOR G = ALPHA/BETA $
STEP5.      TEMP = ALPHA-BETA*G $
STEP6.      IF TEMP NQ 0. $
STEP7.           BEGIN
                 ALPHA = BETA $
                 BETA = TEMP $
                 GOTO STEP4 $
            END  END
STEP8. GAMMA = BETA $
       END
```



PROCEDURES

A procedure is a self-contained routine with a fixed and ordered set of parameters, permanently defined by a procedure declaration and invoked either by a procedure statement or by a function. A procedure, like a closed statement, is a closed subroutine. But, unlike a closed statement, a procedure's parameters allow the data and other environmental elements on which the procedure operates to be expressed when the procedure is executed, rather than when it is compiled. A procedure's parameters are parameters, therefore, because the information they supply the procedure, while fixed for any particular execution of the procedure, may differ from execution to execution.

In classifying procedure parameters, three different dichotomies are useful. Thus, a procedure parameter is either (1) formal or calling, (2) input or output, and (3) value or name. (In consequence, there are eight different types of procedure parameters, from formal input value parameters to output name calling parameters.) These dichotomies may be explained as follows: (1) A formal parameter is a dummy name within the procedure declaration by which the procedure's parametric information is referenced for every execution of the procedure, while a calling parameter is a formula, a variable, or a name within the procedure statement or function by which the procedure's parametric information is expressed for a single execution of the procedure; (2) The information referenced or expressed by an input parameter is provided to the procedure before its execution, while the information referenced or expressed by an output parameter is provided by the procedure after its execution; (3) The information provided by a procedure parameter is either a value, or a name denoting a statement, an array, or a table. The main difference between value and name parameters is that formal value parameters are allocated memory space since they refer to values designated by items declared within the procedure, whereas no memory space is allocated for formal name parameters since they refer to arrays, tables, or statements outside the procedure.

PROCEDURE DECLARATIONS. A procedure declaration is composed of a procedure declaration proper, which declares the procedure's name and lists its formal parameters, followed by an optional list of heading declarations, which describe the information environment peculiar to the procedure, and followed by a statement, which constitutes the body of the procedure.

declaration ≠ PROCedure name$_{\text{of-procedure}}$ ⌊(⌊names'⌋ ⌊= ⌈name ⌊•⌋⌉s'⌋)⌋ $ ⌊declarations⌋ statement

The procedure declaration proper, composed of the PROCedure declarator followed by a name and an optional list of formal parameters enclosed in the (and) brackets and terminated by the $ separator, declares the procedure's name and lists its formal parameters, if any, thus declaring their names, their number, and their order. The dummy names comprising this list of formal parameters are separated by commas and may be divided by a single = separator into formal input parameters on the left and formal output parameters on the right. A procedure may thus have no parameters at all, or it may have both input and output parameters, just input parameters, or just output parameters depending on the presence and position of the = separator. Formal parameters, both input and output, are either value parameters or name parameters and consequently refer either to data values or to the names of statements, arrays, or tables. A formal output parameter that refers to a statement name is suffixed by the . separator.

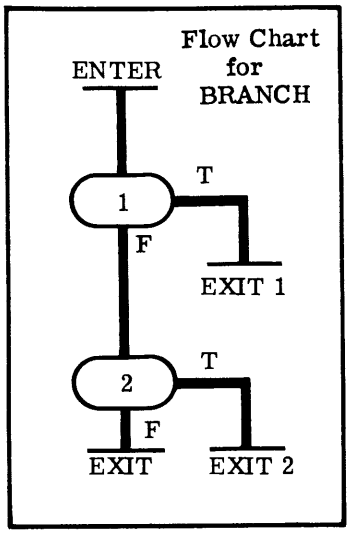
The optional list of heading declarations describe whatever information environment is peculiar to the procedure. Procedure declarations themselves may not be nested, but declarations of any other type may appear in a procedure heading, all describing environmental elements local to the procedure. (Identifiers declared inside a procedure thus do not conflict with identical identifiers declared outside the procedure.) Environmental elements that the procedure shares with the main program consequently must not be redeclared within the procedure, but all of the procedure's formal parameters are considered part of the procedure's local environment and must therefore be declared in the procedure heading, except those formal name parameters referring to statement names. Value parameters must be declared as items, and name parameters must be declared as arrays or tables in order to provide the procedure with a fixed description of their structure. A formal input value parameter can also be used as an output value parameter in the same procedure, but only one item declaration in the procedure heading is necessary to define it.

The following procedure declaration illustrates many of the concepts just discussed. The procedure it describes, SET'DIAGONAL, sets the items on the main diagonal of any 5Ø by 5Ø floating-point matrix to any given numeric value.

```
PROCedure SET'DIAGONAL (VALUE=MATRIX) $
ITEM      VALUE        Floating Rounded $
ARRAY     MATRIX 5Ø 5Ø Floating Rounded $
                       BEGIN
                       FOR I = Ø,1,49 $
                            MATRIX($I,I$) = VALUE $
                       END
```

Notice that VALUE is a formal input value parameter, while MATRIX is a formal output name parameter.

The following procedure, BRANCH, shows how formal output parameters that refer to statement names are used as alternate exits, as graphically illustrated in the accompanying flow diagram.



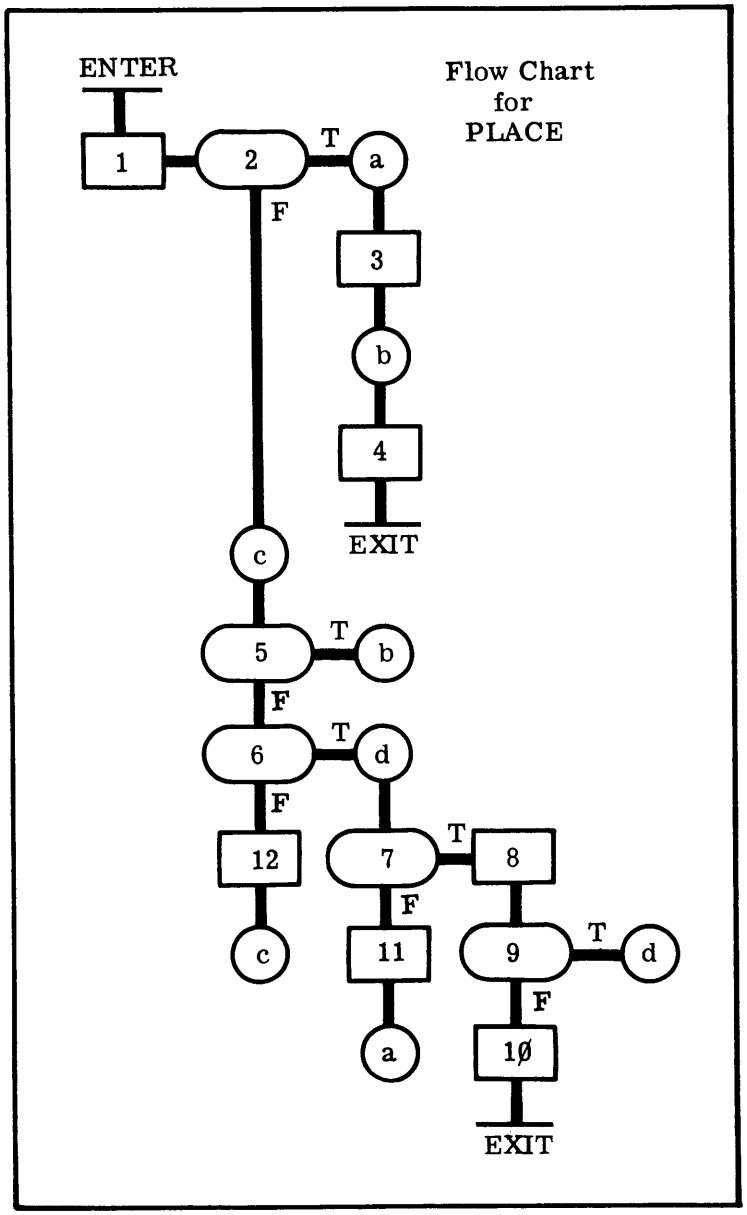

```
PROCedure BRANCH (ALPHA=HIGH.,LOW.) $
ITEM       ALPHA Floating Rounded $
                 BEGIN
               STEP1. IF ALPHA GR ∅. $
                           GOTO HIGH $
               STEP2. IF ALPHA LS ∅. $
                           GOTO  LOW $
                 END
```

Formal input parameters that refer to statement names, on the other hand, usually refer to closed statements and consequently are not alternate exits.

The following procedure declaration declares a typical procedure, PLACE, which is of interest mainly because it performs its searching function much quicker than would an equivalent entry-by-entry search, or even a binary search. Notice that PLACE invokes another procedure via the HASH function. Procedures may invoke other procedures, either through functions or through procedure statements, but they may not invoke themselves, either directly or indirectly.

```
TABLE     WORDS Rigid 1ØØØØ Serial Dense $
                BEGIN
ITEM       WORD Hollerith 6 $
ITEM   OVERFLOW fixed 15 Unsigned $
ITEM      COUNT fixed 15 Unsigned $
                END
PROCedure PLACE (CODE'WORD=CHANNEL,FULL) $''A procedure that places, in
the sense of either find or file, a 6-character code word in a table of
such words and responds with a channel number indexing the table entry
where the code word was either found or filed. An indication is given
if the table is so full that a new code word may not be filed. The HASH
function maps a 6-character code word into a 1Ø-bit integer, a many-to-
one mapping.''
ITEM  CODE'WORD Hollerith 6 $
ITEM    CHANNEL fixed 15 Unsigned $
ITEM       FULL Boolean Preset Ø $
ITEM       NEXT fixed 15 Unsigned Preset 1 $
                BEGIN
         STEP1. FOR N = HASH (CODE'WORD) $
                     BEGIN
         STEP2.      IF WORD($N$) EQ 6H(      ) $''which means that the
entry is vacant''        BEGIN
         STEP3.           WORD($N$) = CODE'WORD $
         STEP4.           CHANNEL = N $
                          RETURN $
                          END
         STEP5.      IF WORD($N$) EQ CODE'WORD $
                          GOTO STEP4 $
         STEP6.      IF OVERFLOW($N$) EQ Ø $''which means that there has
been no overflow from this entry''
                          BEGIN
         STEP7.           IF WORD($NEXT$) NQ 6H(      ) $
                               BEGIN
         STEP8.                NEXT = NEXT+1 $
         STEP9.                IF NEXT LS NENT(WORDS) $
                                    GOTO STEP7 $
         STEP1Ø.               FULL = 1 $
                               RETURN $
                               END
         STEP11.          OVERFLOW($N$) = NEXT $
                          GOTO STEP3 $
                          END
         STEP12.     N = OVERFLOW($N$) $
                     GOTO STEP5 $
                END  END
```

Notice, in the PLACE procedure above, that the WORDS table is part of the environment of both the procedure and the program containing the procedure. The flow chart below shows the rather complicated sequence structure of the PLACE procedure.



FUNCTION DECLARATIONS. A procedure declaration declaring a procedure to specify a function value differs from other procedure declarations in one respect only: it has only one formal output parameter, a value parameter that is taken to be the name of the procedure itself. The procedure declaration for a function therefore lists no formal output parameters since the procedure name itself serves this purpose. And because the procedure name does serve as a formal output value parameter, it must be declared among the procedure's heading declarations as an item, to which the function's value is assigned during the procedure's execution.

To illustrate function declarations, the following procedure declares a Boolean function, NEAR, that specifies True when the floating-point values of its two input parameters are similar in magnitude, and False when they are not.

```
PROCedure NEAR (AA,BB) $
ITEM      NEAR Boolean $
ITEM        AA Floating $
ITEM        BB Floating $
               NEAR = CHAR(AA) EQ CHAR(BB) $
```

The following procedure declaration, defining the CUBER function, shows how a mode declaration can be used to declare all the formal value parameters of a procedure, as long as they are the same type.

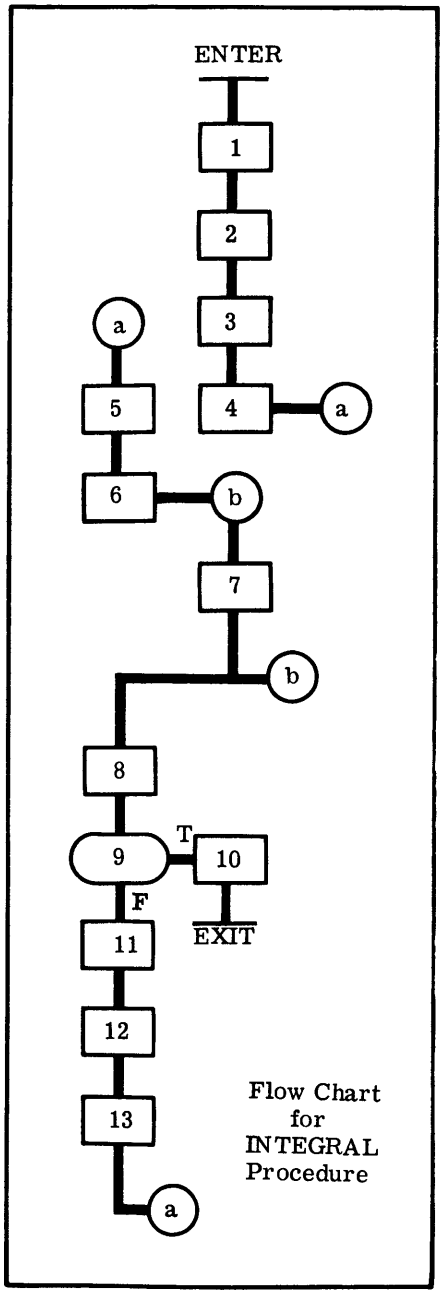
```
PROCedure CUBER (AA,CC) $''This procedure specifies the function value
CUBER such that CUBER**3+AA*CUBER**2+CC EQ Ø., where AA and CC have the
same sign.''
MODE              Floating Rounded $
OVERLAY           AA,CUBER,CC=TØ,T1,T2 $
                  BEGIN
                  TØ    = AA/3. $
                  T1    = TØ**3 $
                  T2    = T1+CC/2. $
                  T1    = (T2**2-T1**2)**.5 $
                  CUBER = (T1-T2)**.3333333333-(T1+T2)**.3333333333-TØ $
                  END
```

In the above procedure, the formal value parameters AA, CUBER, and CC are used as temporary items, and the overlay declaration merely gives them the new names TØ, T1, and T2 to make this explicit.

Some further examples of procedure declarations, which define the functions BINOCO, POLY, INTEGRAL, and ABBREVIATION, are given on the following pages.

```
PROCedure BINOCO (NN,MM) $''This procedure specifies the binomial co-
efficient, BINOCO, the number of combinations of NN things taken MM at
a time = FACTORIAL(NN)/(FACTORIAL(NN-MM)*FACTORIAL(MM)) using the re-
cursion algorithm described by Kenyon in ACM Communications for Oct. 60''
ITEM      BINOCO fixed 48 Unsigned $
ITEM          NN fixed 06 Unsigned 1...50 $
ITEM          MM fixed 06 Unsigned 1...50 $
                 BEGIN
                 BINOCO = 1 $
                 IF 2*MM GR NN $
                      MM = NN-MM $
                 FOR I = 0,1,MM $
                      BINOCO = (NN-I)*BINOCO/(I+1) $
                 END

PROCedure POLY (AA,MM,XX) $''This procedure specifies the function, POLY,
as the value of the polynomial AA($MM$)*XX**MM + AA($MM-1$)*XX**(MM-1) +
... + AA($1$)*XX + AA($0$), where AA is a linear array of coefficients,
MM is the degree of the polynomial, and XX is the variable.''
ITEM      POLY Floating Rounded $
ITEM        MM fixed 10 Unsigned $
ITEM        XX Floating Rounded $
ARRAY  AA 1024 Floating Rounded $
               BEGIN
               POLY = 0. $
               FOR M = MM,-1,0 $
                    POLY = POLY = POLY*XX+AA($M$) $
               END
```



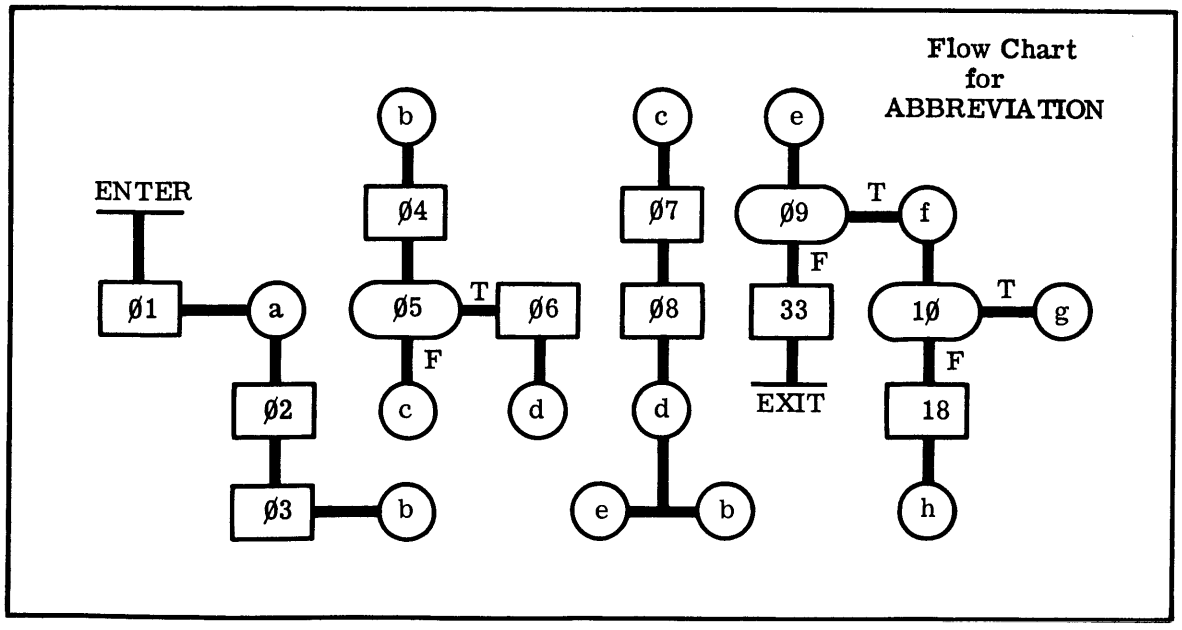
Flow Chart for INTEGRAL Procedure

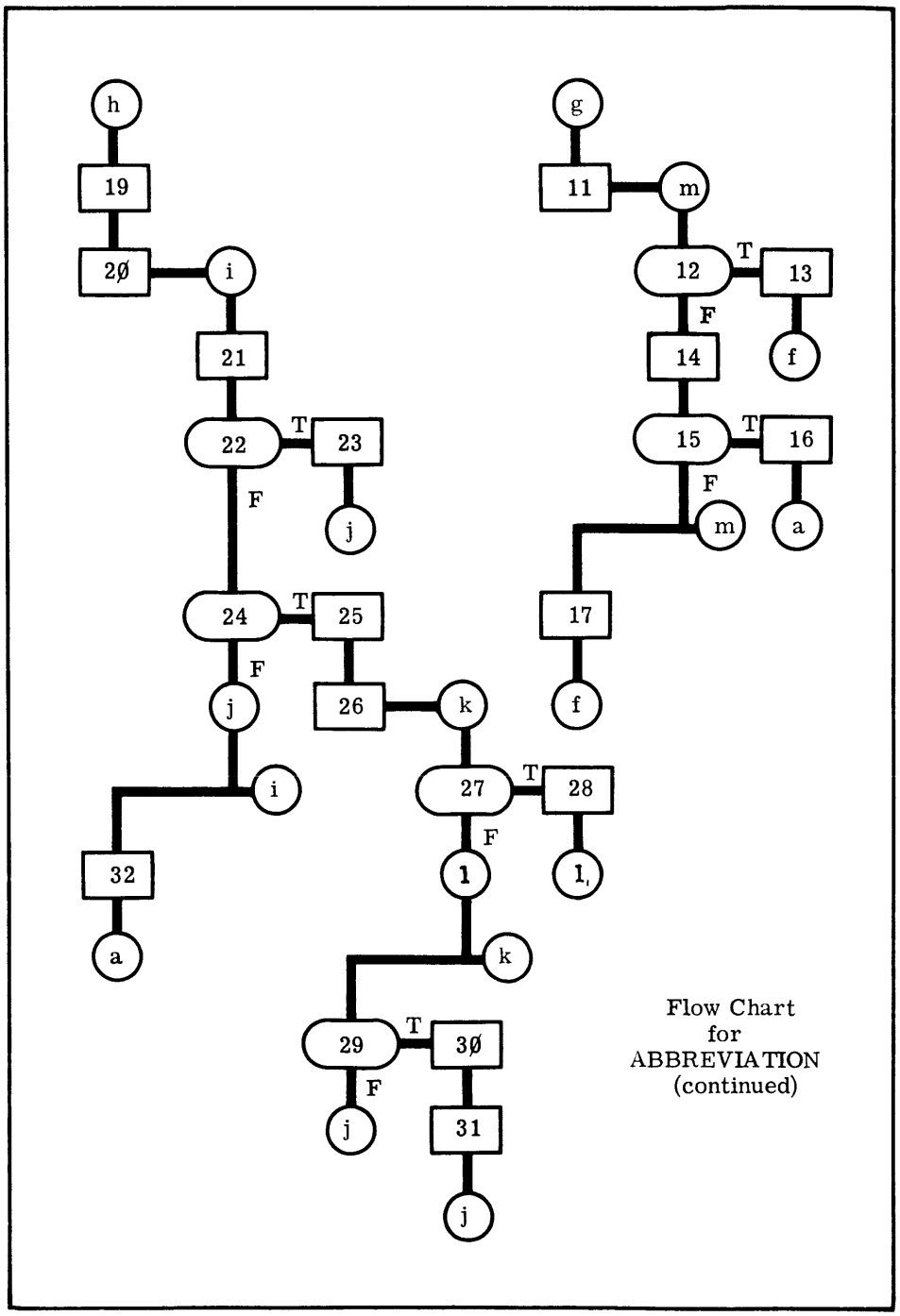
```
PROCedure INTEGRAL (ALPHA,BETA,GAMMA,DELTA) $
''This procedure specifies the INTEGRAL of any
numeric FUNCTION(ALPHA...BETA) by Simpson's
one-third rule.  GAMMA is greater than the
maximum absolute value of FUNCTION(ALPHA...
BETA) and DELTA is the magnitude of the per-
missible difference between two successive
approximations.  The user of this procedure
must fill in the following define declaration
with the name of the function he wishes to
integrate.''
DEFINE FUNCTION ''                          '' $
MODE Floating Rounded $
        BEGIN
STEPØ1. TEMP1 = GAMMA*(BETA-ALPHA) $
STEPØ2. TEMP2 = (BETA-ALPHA)/2. $
STEPØ3. TEMP3 = TEMP2*(FUNCTION(ALPHA)
        + FUNCTION(BETA)) $
STEPØ4. FOR N = 1,N $
              BEGIN
STEPØ5.       TEMP4 = Ø. $
STEPØ6.       FOR K = 1,1,N $
STEPØ7.             TEMP4 = TEMP4+FUNCTION
        (ALPHA+(2*K-1)*TEMP2) $
STEPØ8.       TEMP5 = TEMP3+4.*TEMP2*TEMP4 $
STEPØ9.       IF DELTA GQ (/TEMP5-TEMP1/) $
                    BEGIN
STEP1Ø.             INTEGRAL = TEMP5/3. $
                    RETURN $
                    END
STEP11.       TEMP1 = TEMP5 $
STEP12.       TEMP3 = (TEMP5+TEMP3)/4. $
STEP13.       TEMP2 = TEMP2/2. $
        END  END

PROCedure ABBREVIATION (WORD) $
''This procedure specifies a systematic,
six-letter abbreviation of an English
word of 7...36 letters.  A word of six
or less letters is its own abbreviation.''
ITEM      ABBREVIATION Hollerith 6 $
ITEM              WORD Hollerith 36 $
ITEM            LETTER Hollerith 1 $
```

```
DEFINE FREQUENCY ''V(E) V(T) V(A) V(O) V(N) V(I) V(S) V(R) V(H) V(L)
                   V(D) V(C) V(U) V(M) V(F) V(Y) V(W) V(G) V(P) V(K)
                   V(B) V(V) V(X) V(J) V(Q) V(Z)'' $
ARRAY       CLASS 2 Status V(OTHER) FREQUENCY $
TABLE               Rigid 26 $
                    BEGIN
ITEM        P'CLASS Status V(OTHER) FREQUENCY $  BEGIN FREQUENCY END
ITEM       P'LETTER Hollerith 1 $  BEGIN 1H(E) 1H(T) 1H(A) 1H(O) 1H(N)
                    1H(I) 1H(S) 1H(R) 1H(H) 1H(L) 1H(D) 1H(C) 1H(U) 1H(M)
                    1H(F) 1H(Y) 1H(W) 1H(G) 1H(P) 1H(K) 1H(B) 1H(V) 1H(X)
                    1H(J) 1H(Q) 1H(Z) END
                    END
                    BEGIN
            STEPØ1. FOR J = Ø $
COMPRESS.                   BEGIN
            STEPØ2.         FOR M = 35,-1,Ø $
            STEPØ3.         FOR N = 35,-1 $
                                BEGIN
            STEPØ4.             LETTER = BYTE($M$)(WORD) $
            STEPØ5.             IF LETTER EQ 1H( ) $
            STEPØ6.                 TEST M $
            STEPØ7.             BYTE($M$)(WORD) = 1H( ) $
            STEPØ8.             BYTE($N$)(WORD) = LETTER $
                                END
COMPLETE    STEPØ9.         IF BYTE($Ø,3Ø)(WORD) NQ
3ØH(                                ) $
                                BEGIN
ELIMINATE.  STEP1Ø.             IF J LS 3 $
                                    BEGIN
            STEP11.                 FOR I = 35,-1,1 $
                                        BEGIN
            STEP12.                     IF BYTE($I$)(WORD) EQ 1H( ) $
            STEP13.                         BEGIN
                                            J = J+1 $
                                            GOTO ELIMINATE $
                                            END
            STEP14.                     LETTER = BYTE($I-1$)(WORD) $
            STEP15.                     IF (J EQ Ø AND BYTE($I$)(WORD)
EQ  1H(U) AND LETTER EQ 1H(Q)) OR (J EQ 1 AND BYTE($I$)(WORD) EQ LETTER)
OR (J EQ 2 AND (BYTE($I$)(WORD) EQ 1H(A) OR BYTE($I$)(WORD) EQ 1H(E) OR
BYTE($I$)(WORD) EQ 1H(I) OR BYTE($I$)(WORD) EQ 1H(O) OR BYTE($I$)(WORD)
EQ 1H(U)) AND (LETTER EQ 1H(A) OR LETTER EQ 1H(E) OR LETTER EQ 1H(I) OR
LETTER EQ 1H(O) OR LETTER EQ 1H(U))) $      BEGIN
            STEP16.                         BYTE($I$)(WORD) = 1H( ) $
                                            GOTO COMPRESS $
                                        END  END
            STEP17.                 J = J+1 $  GOTO ELIMINATE $
                                    END
```

```
            STEP18.          CLASS($1$) = V(Z) $
            STEP19.          FOR I = Ø,1,35 $
            STEP2Ø.          FOR K = Ø,1 $
                                  BEGIN
            STEP21.               LETTER = BYTE($I$)(WORD) $
            STEP22.               IF LETTER EQ 1H( ) $
            STEP23.                    TEST I $
            STEP24.               IF K GQ 3 $
                                       BEGIN
CLASSIFY.   STEP25.                    CLASS($Ø$) = V(OTHER) $
            STEP26.                    FOR L = Ø,1,25 $
                                            BEGIN
            STEP27.                         IF LETTER EQ P'LETTER($L$) $
            STEP28.                              CLASS($Ø$) = P'CLASS
($L$) $                                     END
            STEP29.                    IF CLASS($Ø$) LQ CLASS($1$) $
                                            BEGIN
            STEP3Ø.                         CLASS($1$) = CLASS($Ø$) $
            STEP31.                         J = I $
                                  END  END  END
            STEP32.          BYTE($J$)(WORD) = 1H( ) $
                             GOTO COMPRESS $
                             END
ABBREVIATE. STEP33.      ABBREVIATION = BYTE($3Ø,6$)(WORD) $
                    END  END
```





Flow Chart
for
ABBREVIATION
(continued)

PROCEDURE STATEMENTS. To execute the process defined in a procedure declaration, it is necessary to invoke the procedure by a procedure statement (or a function). A procedure statement, which may be thought of as a shorthand description of the process it invokes, has a format similar to that of the procedure declaration proper -- with the PROCedure declarator removed. In other words, a procedure statement is composed of a procedure name followed by an optional list of calling parameters enclosed in the (and) brackets and terminated by the $ separator. The = separator, when it appears, separates input calling parameters on the left from output calling parameters on the right.

statement ≠ name$_{\text{of-procedure}}$ ⌊(⌊⌈formula;name⌉s'⌋ ⌊=⌈variable;

⌈name ⌊.⌋⌉s'⌋)⌋ $

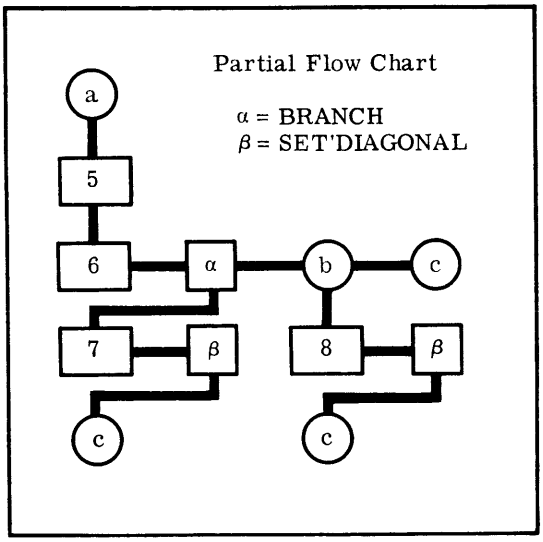
As mentioned before, the information provided a procedure by one of its parameters is either a data value or the name of an array, a table, or a statement. This information, as expressed or denoted by the calling parameters of a procedure statement, is transmitted to the procedure whenever its execution is invoked by the execution of a procedure statement. Input values are specified by calling parameter formulas, output values are designated by calling parameter variables, and both input and output names are directly denoted by calling parameter names. (An output calling parameter statement name must have a . separator after it.)

A procedure statement's calling parameters must correspond exactly to the formal parameters of the synonymous procedure declaration, both in number and in sequence. Calling parameters may not therefore be omitted. In addition, a calling parameter must agree with its corresponding formal parameter -- in data type for value parameters, and in grammatical usage for name parameters. Thus, if a formal value parameter is declared in the procedure heading as a numeric, literal, status, or Boolean item, then the corresponding calling parameter formula or variable must express a numeric, literal, status, or Boolean value. And if a formal name parameter is declared as an array or table, or used as a statement name, then the corresponding calling parameter must be an array name, a table name, or a statement name.

The following statement includes examples of procedure statements that invoke the previously declared procedures SET'DIAGONAL and BRANCH.

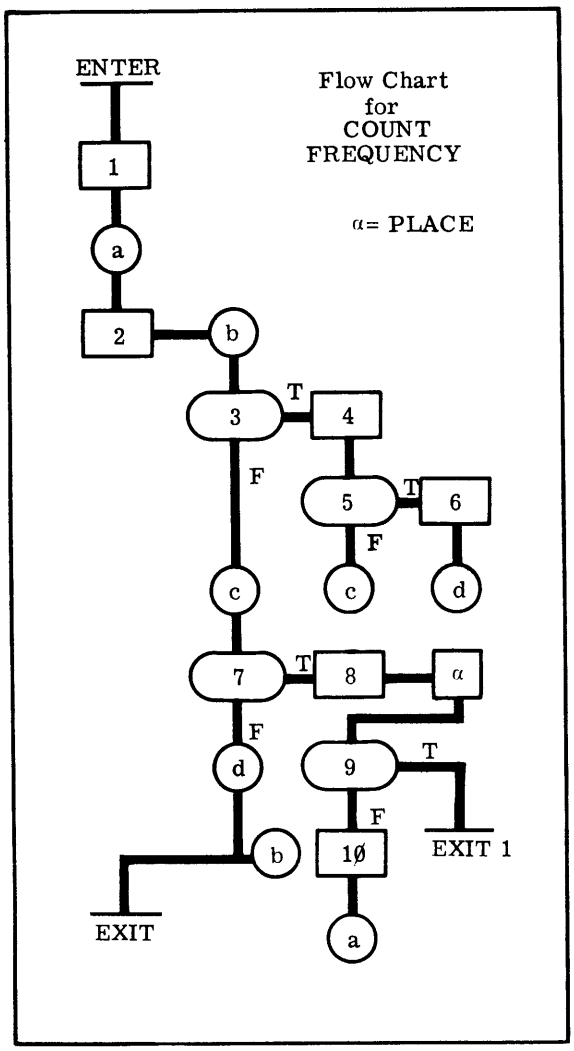
```
                BEGIN
ARRAY COEFFICIENT 5Ø 5Ø Floating Rounded $
ITEM      AVERAGE       Floating Rounded $
          STEP1. AVERAGE = Ø. $
          STEP2. FOR I = Ø,1,49 $
                      BEGIN
          STEP3.      FOR J = Ø,1,49 $
          STEP4.           AVERAGE = AVERAGE+COEFFICIENT($I,J$) $
                      END
          STEP5. AVERAGE = AVERAGE/25ØØ. $
          STEP6. BRANCH (AVERAGE=STEP8.,NEXT'STATEMENT.) $
          STEP7. SET'DIAGONAL (1.=COEFFICIENT) $
                 GOTO NEXT'STATEMENT $
          STEP8. SET'DIAGONAL (AVERAGE=COEFFICIENT) $
                 END
  NEXT'STATEMENT.
```

The above statement computes AVERAGE, the average of all the elements of the COEFFICIENT matrix, and sets those elements on the main diagonal to one if AVERAGE is zero and to the average itself if AVERAGE is greater than zero. The accompanying partial flow chart shows the execution sequence for the more relevant portion of the statement, steps 5 through 8.



To further illustrate procedure statements, consider the following procedure declaration, which contains a procedure statement, STEP8, within its compound body statement.

```
PROCedure COUNT'FREQUENCY (CHARACTER,LENGTH=FULL'EXIT.) $''This procedure,
using the previously declared PLACE procedure and the ABBREVIATION function,
adds to a count of the frequency of occurrence of identical word abbrevi-
ations from a body of English text, provided as a character array of a
given length.  (For this procedure, a word is the first 36 letters of any
unbroken string of letters.)  This procedure will take the FULL'EXIT after
1ØØØØ different abbreviations have been filed by the PLACE procedure.
This procedure also uses a Boolean function, LETTER, which specifies True
only when its one-character input parameter is a Hollerith coded letter.''
ARRAY CHARACTER Ø Hollerith 1 $
ITEM     LENGTH fixed 17 Unsigned 1...1E5 $
ITEM    CHANNEL fixed 15 Unsigned $
ITEM       FULL Boolean $
ITEM       WORD Hollerith 36 $
                BEGIN
         STEP1. FOR C = Ø,1,LENGTH-1 $
         STEP2. FOR W = Ø,1 $
                     BEGIN
         STEP3.      IF LETTER (CHARACTER($C$)) $
                          BEGIN
         STEP4.           BYTE($W$)(WORD) = CHARACTER($C$) $
         STEP5.           IF W LS 35 $
         STEP6.                TEST $
                          END
         STEP7.      IF W GR Ø $
                          BEGIN
         STEP8.           PLACE (ABBREVIATION (WORD) = CHANNEL,FULL) $
         STEP9.           IF FULL $  GOTO FULL'EXIT $
         STEP1Ø.          COUNT($CHANNEL$) = COUNT($CHANNEL$)+1 $
                          GOTO STEP2 $
                END  END  END
```



The accompanying flow chart shows the statement execution sequence for COUNT'FREQUENCY, and the following statement -- which reads a deck of punched cards with the aid of a READ procedure, counts abbreviation frequencies, and prints them out with the aid of a special PRINT procedure -- shows how COUNT'FREQUENCY might be employed.

```
    BEGIN
ARRAY CARD 8ø Hollerith 1 $
S1. FOR I = ALL (WORDS) $
S2.       BEGIN
    WORD($I$) = 6H(       ) $
    OVERFLOW($I$) = ø $
    COUNT($I$) = ø $
          END
S3. READ (=CARD,S5.,S6.) $
S4. COUNT'FREQUENCY (CARD,
    8ø = S5.) $  GOTO S3 $
S5. PRINT(WORDS=S6.) $
S6. STOP $
    END
```

In the above routine: statements S1 and S2 initialize the WORDS table; S3 reads a card, going to S5 if there are no more cards and to S6 if the read operation is unsuccessful; S4 counts abbreviation frequencies; S5 converts the WORDS table to a legible Hollerith format and prints it out on the line printer, going to S6 if the print operation is unsuccessful; and S6 stops the computer.

To better understand the relationship between a procedure's calling parameters and its formal parameters, consider the following procedure declaration along with an equivalent closed statement.

```
PROCedure PØ (P1=P2,P3) $
ITEM      P1 Floating $
ITEM      P2 Floating $
ARRAY     P3 1ØØ Floating $
BEGIN
P2 = Ø. $
FOR I = Ø,1,99 $
     BEGIN
     P3($I$) = P3($I$)/P1 $
     P2 = P2+P3($I$) $
     END
P2 = P2/1ØØ. $
END
```

```
CLOSE     PØ $
BEGIN
ITEM      P1 Floating $
ITEM      P2 Floating $
ARRAY     P3 1ØØ Floating $
P2 = Ø. $
FOR I = Ø,1,99 $
     BEGIN
     P3($I$) = P3($I$)/P1 $
     P2 = P2+P3($I$) $
     END
P2 = P2/1ØØ. $
END
```

Both procedure and closed statement perform the same ostensive function, that of dividing vector P3 by value P1 and assigning the norm (component average) of the resulting vector to P2. Their actual effects, however, differ slightly in several minor details: (1) no memory space is allocated the procedure's P3 array, whereas the closed statement's P3 array is, of course, allocated memory space; (2) the identifiers P1, P2, and P3 naming the procedure's formal parameters are local to the procedure, whereas the same identifiers in the closed statement are defined for the listed remainder of the program; (3) the closed statement, being a statement, may be part of a compound statement with active subscripts that would also be active over the closed statement, while a procedure declaration, being a declaration, cannot be within a subscript's range of activity even if it is listed within such a compound statement. (A procedure must activate its own subscripts; the values of program-activated subscripts can only be imparted to a procedure as calling parameters.)

The following pair of statements, by invoking the process described by both the procedure declaration and the closed statement above, produce exactly the same operational effect as far as ALPHA, BETA, and GAMMA are concerned. The only difference is that, in the closed statement call on the right, the declared overlay of GAMMA and P3 is permanent, while the overlaying effect implied by the procedure's call-by-name of GAMMA is temporary and only lasts for the duration of the procedure's execution.

```
PØ (ALPHA+7.=BETA,GAMMA) $
```

```
BEGIN
OVERLAY GAMMA = P3 $
P1 = ALPHA+7. $
GOTO PØ $
BETA = P2 $
END
```

The mechanism for assigning formal parameter values generated by the procedure statement on the left is similar in effect (if not in operation) to the assignment statements before and after the GOTO invoking the closed statement PØ. Output value calling parameters are thus assigned the values designated by their corresponding formal parameters immediately after the completion of the procedure's execution and before any other statement is executed. A procedure's execution is "completed" by the execution of its last listed statement, or by a RETURN statement, or by a GOTO statement referring to an alternate exit (i.e., one of the procedure's formal output, statement name parameters). It should be noted that a GOTO statement transferring execution control to any other statement outside the procedure does not in this sense complete the procedure's execution, though it may well terminate it.

THE REMQUO PROCEDURE. A JOVIAL procedure can be considered an extension to the language and, in this sense, JOVIAL is an open-ended language since any number of procedures can be added to the COMPOOL's library of procedures. These library procedures then become, as far as the programmer is concerned, part of the language itself. The REMQUO (REMainderQUOtient) procedure is one such procedure and, since it remedies a basic lack in the language, it is a common procedure available to all JOVIAL programmers, who may think of it as an intrinsic part of the language.

REMQUO is used when, in performing integer division, both quotient and remainder are required, which would otherwise be available only after a much less efficient process of repeated subtractions. Like any other procedure, REMQUO is invoked by a procedure statement, which has the format:

```
REMQUO (DIVIDEND,DIVISOR=QUOTIENT,REMAINDER) $
```

In other words, the following pair of rather simple-minded, compound statements are operationally equivalent:

```
BEGIN                                  BEGIN
NUM = 7 $                              NUM = 7 $
DEN = 3 $                              DEN = 3 $
REMQUO (NUM,DEN=QUO,REM) $             QUO = 2 $
END                                    REM = 1 $
                                       END
```

Although REMQUO's parameters are formally integers, both dividend and divisor can be specified by any mono-valued numeric formula, and quotient and remainder can be designated by any numeric variable, for truncation and conversion between fixed and floating-point representation are, of course, automatic -- as with any numeric parameter in other procedures.

The following procedure, which employs REMQUO, will serve to illustrate its use.

```
PROCedure STERLING (CENTS=POUNDS,SHILLINGS,PENCE) $''STERLING converts
up to a million dollars of American money (in cents) to British money
(in pounds, shillings, and pence). Twelve pence make one shilling and
twenty shillings make one pound. The current exchange rate ($2.81 per
pound) is given in a define declaration, which must be altered if the
rate should change.''
ITEM          CENTS fixed 27 Unsigned Ø...1E8 $
ITEM         POUNDS fixed 19 Unsigned $
ITEM     SHILLINGS fixed Ø5 Unsigned $
ITEM         PENCE fixed Ø4 Unsigned $
DEFINE        RATE ''281'' $
                   BEGIN
                   REMQUO (CENTS,RATE=POUNDS,CENTS) $
                   REMQUO (CENT,RATE/2Ø=SHILLINGS,CENTS) $
                   PENCE = CENTS*24Ø/RATE $
                   END
```

EXERCISE (Procedures)

(a) The following three procedures presume to compute the factorial of a number between Ø and 15. Examine them, verify this, and comment on their various methods of operation.

```
PROCedure FACTORIAL (NUMBER) $
ITEM      FACTORIAL fixed 41 Unsigned 1...1307674368ØØØ $
ITEM      NUMBER fixed 4 Unsigned Ø...15 $
ARRAY     FACTORIALS 16 fixed 41 Unsigned $
               BEGIN          1 ''ØØ''
                              1 ''Ø1''
                              2 ''Ø2''
                              6 ''Ø3''
                             24 ''Ø4''
                            12Ø ''Ø5''
                            72Ø ''Ø6''
                          5Ø4Ø ''Ø7''
                         4Ø32Ø ''Ø8''
                        362880 ''Ø9''
                       3628ØØ ''1Ø''
                      3991680Ø ''11''
                     479ØØ16ØØ ''12''
                    6227Ø2Ø8ØØ ''13''
                   87178291 2ØØ ''14''
                 13Ø767436 8ØØØ ''15''
                 END
                 FACTORIAL = FACTORIALS($NUMBER$) $
```

```
PROCedure FACTORIAL (NUMBER) $
ITEM      FACTORIAL fixed 41 Unsigned 1...13Ø7674368ØØØ $
ITEM      NUMBER fixed 4 Unsigned Ø...15 $
          BEGIN
          FACTORIAL = 1 $
          FOR N = 1,1,NUMBER $
               FACTORIAL = N*FACTORIAL $
          END

PROCedure FACTORIAL (NUMBER) $
ITEM      FACTORIAL fixed 41 Unsigned 1...13Ø7674368ØØØ $
ITEM      NUMBER fixed 4 Unsigned Ø...15 $
          BEGIN
          IF NUMBER EQ Ø $
               BEGIN
               FACTORIAL = 1 $
               RETURN $
               END
          FACTORIAL = NUMBER*FACTORIAL(NUMBER-1) $
          END
```

(b) The ABBREVIATION function was declared as a procedure beginning at the bottom of page 167. Choose three English words of at least ten letters and determine what abbreviation the procedure would map them into.

(c) Write a JOVIAL procedure to reverse the order of characters in any linear array of 1-character literal values. Thus, if "MEMORANDUM" is input to this procedure, its output should be "MUDNAROMEM".

(d) The following parameterless procedure is meant to sort a table's entries, by KEY item, into ascending order. Examine it carefully and suggest some improvements.

```
PROCedure QUIK'SORT $
DEFINE    KEY ''To be filled in by the procedure's user with the name
               of a table item.'' $
               BEGIN
               FOR M = NENT(KEY)-1,-(M+1)/2,Ø $
                    BEGIN
                    FOR J = Ø,1,NENT(KEY)-(M+1) $
                         BEGIN
                         FOR I = J,-M,Ø $
                              BEGIN
                              IF KEY($I$) GR KEY($I+M$) $
                                   KEY($I$) == KEY($I+M$) $
               END  END  END  END
```

(e) Integer numbers may be expressed in English with the following vocabulary: ONE; TWO; THREE; FOUR; FIVE; SIX; SEVEN; EIGHT; NINE; TEN; ELEVEN; TWELVE; THIRTEEN; FOURTEEN; FIFTEEN; SIXTEEN; SEVENTEEN; EIGHTEEN; NINETEEN; TWENTY; THIRTY; FORTY; FIFTY; SIXTY; SEVENTY; EIGHTY; NINETY; HUNDRED; THOUSAND; MILLION. Examples are: THREE HUNDRED NINETEEN THOUSAND TWENTY FOUR; NINE HUNDRED NINETY NINE MILLION NINE HUNDRED NINETY NINE THOUSAND NINE HUNDRED NINETY NINE. Write a JOVIAL procedure to evaluate numbers expressed in this fashion.

SWITCHES

A switch is a routine for computing a statement name and, thus, for deciding among many alternate sequences of operation. As a closed statement or procedure is useful when the same computation must be done at many places in a program, so a switch is useful when the same decision must be made at many places in the program. By presetting switches at the start of program execution, the program's main logical flow may also be preset, adapting its operation for a particular pattern of initial input data.

A switch computes a statement name, and a sequential formula specifies a statement name -- either directly, by name; or indirectly, by invoking a switch. Switches are, ultimately, always invoked by GOTO statements, again, either directly, or indirectly through other switches.

sequential-formula ≠ name$_{\text{of-statement}}$; ⌈name$_{\text{of-switch}}$ ⌊($ index $)⌋⌉

statement ≠ GOTO sequential-formula$_{\text{specifying-next-statement}}$ $

A switch call invoking a switch consists of the name of the switch subscripted by an index, which may be omitted if the switch type allows. The switch name refers to the switch declaration defining it, which lists a set of sequential formulas specifying the statement names that the switch call itself may specify. (Since these statement names may themselves be specified by switch calls, the evaluation of a switch call is obviously a recursive process.) An index subscripting a switch name serves either directly to index the list of sequential formulas given in the switch declaration, or to index an item, named in the switch declaration, whose value is then used in comparison with a list of constants to determine which sequential formula in the list is to specify the switch's statement-name value.

As the preceding paragraph implied, there are two kinds of switches -- indexed switches, and item switches.

INDEXED SWITCHES. The statement names that may be computed by an indexed switch are specified as a list of sequential formulas in the switch declaration. Any position in this list may be empty, thus effectively specifying the statement listed after the switch invoking GOTO statement. The declaration for an indexed switch is composed of the SWITCH declarator followed by a name, the = separator, a list of optional sequential formulas separated by commas and enclosed in the (and) brackets, and terminated by the $ separator.

declaration ≠ SWITCH $\text{name}_{\text{of-switch}}$ = (⌊sequential-formula⌋s') $

An n-position list of sequential formulas is indexed in the switch call by a one-component index subscripting the switch name. This index may range in value from ∅ to n-1.

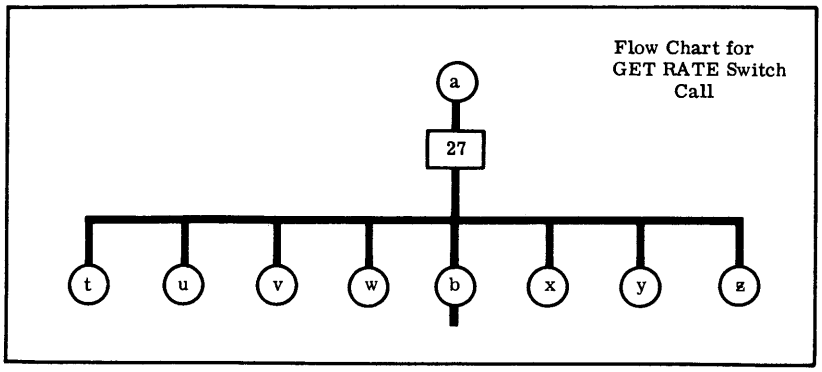
The following simple example will illustrate the declaration and call of an indexed switch:

SWITCH GET'RATE = (PLAN∅,PLAN1,PLAN2,PLAN3, ,PLAN5,PLAN6,PLAN7) $

The call for this switch might be:

STEP27. GOTO GET'RATE(I) $

so that when I designates ∅, the statement named PLAN∅ is executed next, and when I designates 1, the statement named PLAN1 is executed next, and so on, except that when I designates 4, the GOTO statement above has no operational effect whatever and merely transfers execution control to the next statement listed, since position 4 in the switch declaration's list of sequential formulas is empty. Notice also that since this list has eight positions (even though one is empty), the subscript I may only range from ∅ through 7 in value. A value outside this range not only makes the effect of the switch call undefinable, but is a serious program error as well. This switch-invoking statement, STEP27 above, would appear in a flow chart as follows:



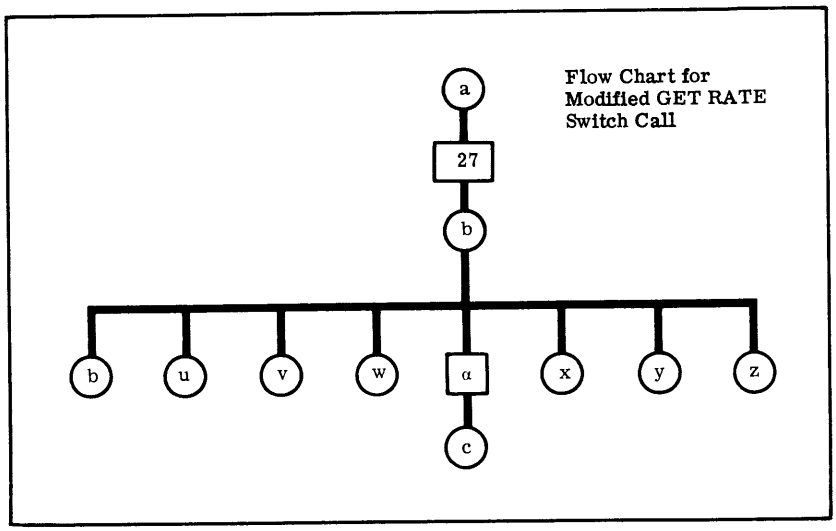
The following example, a more complicated version of the previous one, illustrates two additional features of switches, namely, that the list of sequential formulas in a switch declaration may contain both switch calls, and closed statement names.

```
SWITCH GET'RATE = (GET'RATE($DAY$),PLAN1,PLAN2,PLAN3,FIND'RATE,PLAN5,
                   PLAN6,PLAN7) $
```

In the above declaration, FIND'RATE is presumed to be the name of a closed statement. Notice that the list of sequential formulas in a switch declaration may include not only switch calls, but switch calls for the switch being declared. This, of course, raises the possibility of an infinite loop caused by a circular switch call, as would be the case, for example, if both I and DAY designated zero when the following GOTO statement was executed.

```
STEP27. GOTO GET'RATE($I$) $
```

This switch-invoking GOTO statement would appear in a flow chart as follows:



which graphically re-emphasizes the fact that a closed statement's normal successor is always the statement listed after the GOTO statement invoking it, even if it is indirectly invoked through one or more switches.

ITEM SWITCHES. The statement names that may be computed by an item switch are specified in a list of constant = sequential-formula pairs within the switch declaration, which is composed of the SWITCH declarator followed by the name of the switch, the name of an item enclosed in the (and) brackets, the = separator, a list of constant = sequential-formula pairs separated by commas and enclosed in the (and) brackets, and terminated by the $ separator.

declaration ≢ SWITCH name$_{\text{of-switch}}$ (name$_{\text{of-item}}$) = (⌈constant = sequential-formula⌉s') $

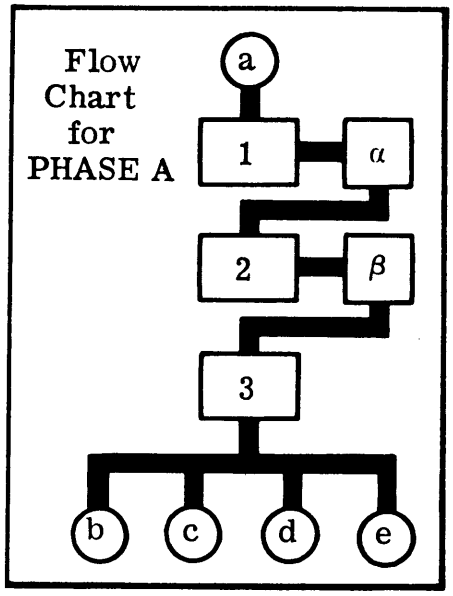
The item name given in the switch declaration and the index (if any) subscripting the switch name in the call together designate an item value. This value selects from the declared list of sequential formulas the one paired with the first listed constant that denotes a value equal to it. If no constant equal to this item value is listed in the declaration, then the switch effectively specifies the statement listed after the switch-invoking GOTO statement.

As an example of an item switch, consider the following pair of declarations, which might have been taken from the JOVIAL compiler itself.

```
ITEM SYMBOL'TYPE Status V(OTHER)
     V(DELIMITER) V(IDENTIFIER)
     V(CONSTANT) $
SWITCH PROCESS'SYMBOL (SYMBOL'TYPE)
     = (V(OTHER)=PROCESS'ERROR,
     V(DELIMITER)=PROCESS'DELIMITER,
     V(IDENTIFIER)=PROCESS'IDENTIFIER,
     V(CONSTANT)=PROCESS'CONSTANT) $
```

The call for the above switch is in the third GOTO statement below. The other two invoke closed statements, as may be seen from the accompanying flow chart.

```
       BEGIN PHASE'A
STEP1. GOTO GET'SYMBOL $
STEP2. GOTO DETERMINE'SYMBOL'TYPE $
STEP3. GOTO PROCESS'SYMBOL $
       END''PHASE'A''
```



INPUT/OUTPUT AND FILES

Many data storage devices impose accessing restrictions in that inserting or obtaining a particular value may involve the transfer of an entire block of data. Such devices are termed "external" storage devices, as contrasted with the "internal" memory of the computer. To allow a reasonably efficient description of input/output processes, therefore, all data entering or leaving the computer's internal memory is organized into files. A file is thus a body of data contained in some external storage device, such as punched cards, and magnetic tape or drums.

FILE DECLARATIONS. Data processed by a digital computer falls into two major categories: data that is contained in the computer's internal

memory and is organized into items, arrays, and tables; and data that is contained in external storage devices and is organized into files and enters or leaves the internal memory of the computer. The operation of any data processing system is marked by a flow of data into the computer and a flow of data out of the computer, for the basic function of any such system is the creation, maintenance, and manipulation of files of data.

A file consists of a string of individual machine language symbols called records. A file of length k may therefore be considered as a k-component vector, arranged as follows:

$$p(\emptyset) \quad R_{\emptyset} \quad p(1) \quad R_1 \quad p(2) \quad R_2 \quad \cdots \quad p(k-1) \quad R_{k-1} \quad p(k)$$

where the R's are records and the p's are partitions* separating the records. Partitions may be interpreted

$p(k)$ = end of file partition; $p(n < k)$ = end of record partition.

Each record in a file is itself a string, either of bits or of six-bit, Hollerith-coded bytes. The records of a file are either all binary or all Hollerith, and they are generally similar to each other in size, content, and format. (When differing records are organized into a file, the programmer must provide for distinguishing between them.) A record, then is a single, usually composite, machine symbol, which may represent an entire block of values when stored in the computer's memory, but which has no internal structure whatever when stored in the file.

A file declaration is composed of the FILE declarator followed by the name of the file, either the Binary or Hollerith type descriptor, the estimated maximum number of records, either the Variable or Rigid record length descriptor, the estimated maximum number of bits or bytes per record, a list of status constants, the name of the storage device containing the file, and finally, the $ separator.

---

* The exact nature of the partitions between records is left undefined. In general, a partition separating one record from the next may result from the operation of the external storage device containing the file, or it may result from the operation of the compiler-created input/output routine processing the file.

declaration ≠ FILE name$_{\text{of-file}}$ Binary;Hollerith number$_{\text{of-records}}$ Variable;Rigid$_{\text{record-length}}$ number$_{\text{of-bits-or-bytes-per-record}}$ status-constants name$_{\text{of-storage-device}}$ $

The status constants listed in the file declaration are associated with the file name and denote the possible states of the storage device containing the file*. File status may thus be determined with a relational Boolean formula wherein the file name is considered as a status variable that is automatically updated prior to comparison according to the current state of the file's storage device.

boolean-formula ≠ name$_{\text{of-file}}$ EQ;GR;GQ;LQ;LS;NQ status-constant

A file name may also be substituted for an item name in the declaration of an item switch, so that file status may also be determined by a GOTO statement invoking such a switch.

declaration ≠ SWITCH name$_{\text{of-switch}}$ (name$_{\text{of-file}}$) = (⌈status-constant = sequential-formula⌉s') $

---

* For purposes of this manual, it will be assumed that storage devices have four possible states, in order:

  state ∅: storage device is not ready, either because it has not been connected to the computer or because the end-of-file partition has been encountered.

  state 1: storage device has transmitted a record or is ready to transmit a record.

  state 2: storage device is busy transmitting a record.

  state 3: storage device is unsuccessful in transmitting a record due to an uncorrectable error.

  It must be realized, however, that the names of storage devices and the number and meaning of their possible states are computer-dependent, so that anyone wishing to declare a workable file must refer to the pertinent documentation for a particular JOVIAL compiler.

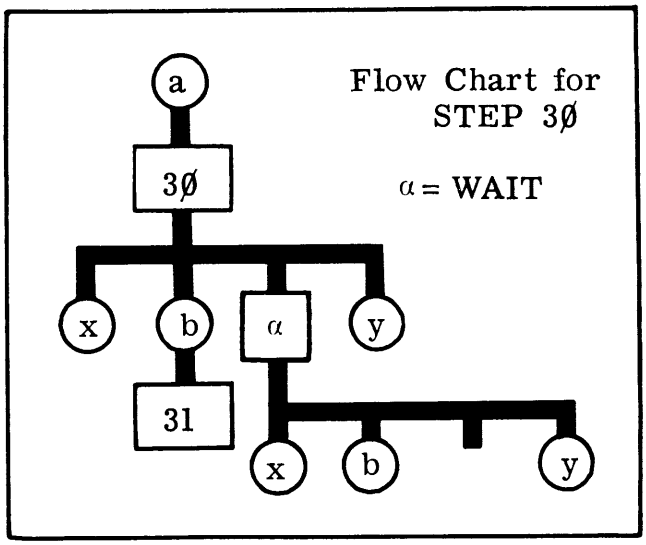
The following example illustrates file declarations and the mechanisms for determining file status.

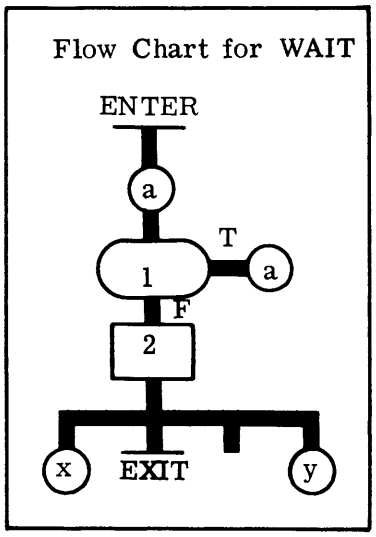
```
FILE INVENTORY Binary 1ØØØØØ Rigid 5ØØ V(UNREADY) V(READY) V(BUSY)
               V(ERROR) TAPE'A $
SWITCH CHECK'INVENTORY'FILE (INVENTORY) = (V( UNREADY )=PROCESS'FILE'END,
               V(BUSY)=WAIT, V(ERROR)=PROCESS'ERROR) $
CLOSE   WAIT $ BEGIN
        STEP1. IF INVENTORY EQ V(BUSY) $ GOTO STEP1 $
        STEP2. GOTO CHECK'INVENTORY'FILE $
               END
```

The above file declaration declares a binary INVENTORY file of no more than one hundred thousand, 5ØØ-bit records, each containing information on a single article in stock. The file is contained on a reel of magnetic tape, mounted on a tape drive with the symbolic name TAPE'A (which presumably has meaning to some JOVIAL compiler). Each of the states of the tape drive is assigned its own arbitrary but mnemonic status constant, and these are used by the CHECK'INVENTORY'FILE switch in determining file status. The statement

```
STEP3Ø. GOTO CHECK'INVENTORY'FILE $
STEP31.
```

will transfer execution control to PROCESS'FILE'END if the file is not ready, to PROCESS'ERROR if the file has encountered an uncorrectable data transmission error, to STEP31 if the file is ready, and to the closed statement WAIT if the file is busy. The closed statement, WAIT, will re-execute its STEP1 until the file is no longer busy, when it will transfer execution control, via the CHECK'INVENTORY'FILE switch, to PROCESS'FILE'END, PROCESS'ERROR, or STEP31. The following pair of flow charts show the entire, involved, execution sequence.





POSITIONING, AND READING AND WRITING FILES. A JOVIAL file is a self-indexing storage device, meaning that the record available for transfer to or from the file depends on the file's current position. The POSition functional modifier, operating on the name of an active* file, is a numeric variable designating a positive, integral value that determines, or is determined by, the current position of the file.

variable$_{\text{of-numeric-type}}$ ≠ POSition (name$_{\text{of-active-file}}$)

For example, if record 3 is currently available for transfer to or from the INVENTORY file, then that file is positioned at partition p(3) and the value designated by

POSition (INVENTORY)

is three. File position, for a file of k records, ranges from ∅ (indicating "rewound") through k (indicating "end-of-file") and, where the characteristics of the storage device allow, file position, as a variable, may be altered by the assignment of a value within this range. Thus,

POSition (INVENTORY) = POSition (INVENTORY) - 1 $

"backspaces" the file, while

POSition (INVENTORY) = ∅ $

"rewinds" the file, and

POSition (INVENTORY) = N $

moves the file to an arbitrary position specified by the subscript, N. Any file for which such a general positioning operation is to be avoided as inefficient (e.g., tape) or impossible (e.g., cards, printer) is called a serial, as opposed to an addressable, file.

The position of a file is also affected by the transfer of a record to or from the file; both a read operation and a write operation increment file position by one.

---

* An active file is one that has been "activated" by the execution of an OPEN INPUT or an OPEN OUTPUT statement, as described in the following sections.

A read operation moves a record from a file into the computer's internal memory so as to represent a block of values; a write operation moves, as a record, the machine symbol representing a block of values (or the value denoted by a constant) from the computer's internal memory out to the file. In either case, the block of values may be designated by a single variable, by an entire array or table, or by a consecutive set of table entries.

block ≠ variable;name$_{\text{of-array}}$;⌈name$_{\text{of-table}}$ ⌊($ index$_{\text{specifying-first-entry}}$ ⌊... index$_{\text{specifying-last-entry}}$⌋ $)⌉

Some examples designating blocks of values are given below:

ALPHA(I)

BYTE($I,8Ø$)(LINE)

MATRIX

PAYROLL($I...I+31$)

A record is a string of bits or bytes with no other explicit structure, and its only implicit structure is supplied by the item, array, or table from which a block of values is designated for transfer to or from the file. Thus, reading and writing are just data transfers, and no editing or conversion occurs, except that required for converting from punched card code to six-bit Hollerith code. A read operation transfers just the bits or bytes of the record, to the maximum designated for the input block, whereas a write operation transfers just the bits or bytes specified for the output block. Consequently, a read operation is terminated either when the entire block of values has been represented by the bits or bytes of the record, or when the last bit or byte of the record has been transferred into the computer's internal memory. A write operation, on the other hand, is terminated only when all the bits or bytes representing the block of values have been transferred, as a record, out to the file.

In general, an active file (i.e., one that has been "opened") may be positioned, read, and written. Some file characteristics, however, occasionally preclude some of these operations. For example: some files are read-only files while others are write-only files; and some files are serial files that may not easily be positioned while others are

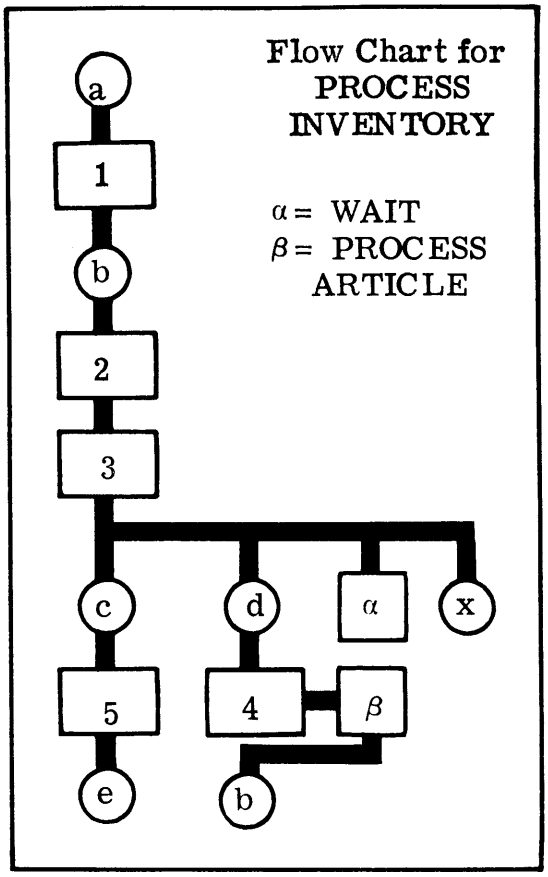
addressable files that can be positioned. The end of a file is indicated by an end-of-file partition. In some files, notably rigid-record-length addressable files and all input files, record partitions are predetermined, so that a read or write operation initiated at the end-of-file partition will (at least in the examples given in this manual) cause the file to become "unready". In other files, however, notably serial output files and variable-record-length addressable output files, end-of-record partitions are created by writing the file and the end-of-file partition is created by deactivating the file. In such files, no records may exist after the last one written, so that a position or read operation beyond this point would result in an error.

The file characteristics mentioned in the paragraph above are listed here.

a. Read-only files; Write-only files; Read-write files.
b. Input files; Output files; Input and output files.
c. Serial files; Addressable files.
d. Variable record length files; Rigid record length files.

Most of these file characteristics depend on the particular storage device containing the file and must, obviously, be taken into account when writing file processing algorithms in JOVIAL.

INPUT STATEMENTS. A file may be read, one record at a time, by the execution of a sequence of INPUT statements. The first statement executed in such a sequence must be an OPEN INPUT statement, which activates the file, and the last statement executed must be a SHUT INPUT statement, which deactivates the file. Before discussing the grammar of INPUT statements, consider the following, almost self-explanatory example, which processes the previously declared INVENTORY file using the CHECK'INVENTORY'FILE switch, a one-entry table, ARTICLE, describing the record format of the file, and a closed statement, PROCESS'ARTICLE, which processes the table and, thus, an INVENTORY record.




```
        BEGIN PROCESS'INVENTORY.
STEP1.  OPEN INPUT INVENTORY $
STEP2.  INPUT INVENTORY ARTICLE $
STEP3.  GOTO CHECK'INVENTORY'FILE $
STEP4.  GOTO PROCESS'ARTICLE $
        GOTO STEP2 $
PROCESS'FILE'END.
STEP5.  SHUT INPUT INVENTORY $
        END''PROCESS'INVENTORY''
```

In the above statement, STEP1 "activates" the INVENTORY file, determining whether it is available for reading -- perhaps whether a reel of magnetic tape with the correct identification has been mounted on the proper tape drive. STEP1 also "rewinds" the file, positioning it so that the first record is ready for transfer. STEP2 initiates a read operation, which will transfer a record from the INVENTORY file to the ARTICLE table and increment the position of the file by one. By invoking the CHECK'INVENTORY'FILE switch, STEP3 will "wait" if the file is "busy", execute an error routine if a data transmission error has occurred, execute STEP5 if the file is "unready" (indicating a read was attempted from the end-of-file position), and execute STEP4 if and when the file is "ready" (indicating the previous record has been successfully transferred). STEP4 "processes" the record just read into the ARTICLE table, perhaps computing a statistical summary of stock shipments, and then returns to execute STEP2 again. And finally, STEP5 "deactivates" the file, releasing the tape drive for possible other use.

An OPEN INPUT statement is composed of the OPEN file operator followed by the INPUT file operator, a file name, the optional designation of a block of values, and terminated by the $ separator.

statement ∉ OPEN INPUT name$_{\text{of-inactive-file}}$ ⌊block⌋ $

An OPEN INPUT statement activates an inactive file and positions it to zero. If a block of values is designated, it also initiates a read operation that will transfer the first record from the file into the computer's internal memory to represent the block of values, thus incrementing the file's position by one.

An INPUT statement is composed of the INPUT file operator followed by a file name, the designation of a block of values, and terminated by the $ separator.

statement ∉ INPUT name$_{\text{of-active-file}}$ block $

An INPUT statement initiates a read operation on an active file that will transfer a record from the file into the computer's internal memory to represent the designated block of values, thus incrementing the file's position by one.

A SHUT INPUT statement is composed of the SHUT file operator followed by the INPUT file operator, a file name, the optional designation of a block of values, and terminated by the $ separator.

statement ∉ SHUT INPUT name$_{\text{of-active-file}}$ ⌊block⌋ $

A SHUT INPUT statement deactivates an active file. If a block of values is designated, it also initiates a final read operation that will, prior to the deactivation of the file, transfer a record from the file into the computer's internal memory to represent the designated block of values.

As another example of reading a file with a sequence of INPUT statements, consider the following statement, which reads a deck of cards punched with the following sales-slip data from a chain of department stores, and updates the accompanying summaries:

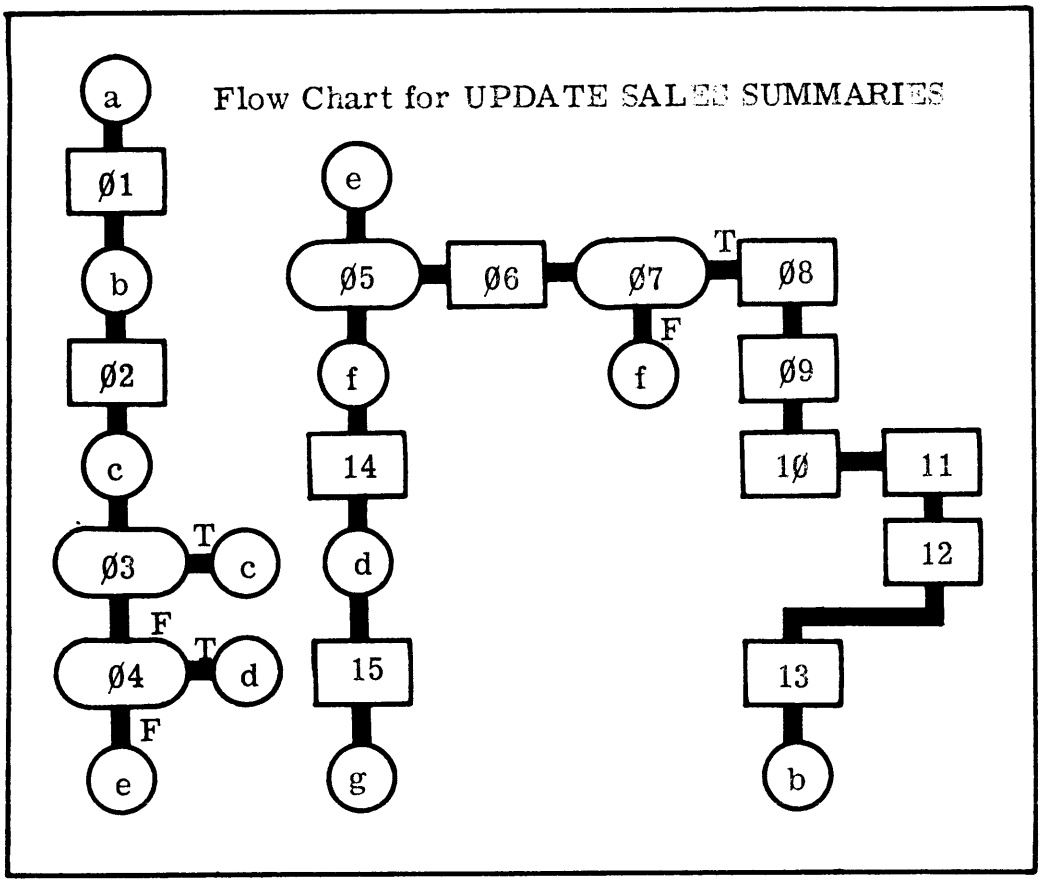
Clerk number	columns ø1...ø4	For each sales clerk
Department number	columns ø6...ø9	Volume of sales
Article number	columns 11...15	Number of sales
Quantity sold	columns 17...2ø	Gross commissions
Price (dollars)	columns 23...26	For each sales department
Price (cents)	columns 28...29	Volume of sales
		Number of sales
		For each article stocked
		Number sold

The statement utilizes a function, NUMBER, which converts a Hollerith coded decimal number of up to ten digits, to a binary numeric value (e.g., NUMBER (9H(999999999)) EQ 999999999).

```
                    BEGIN UPDATE'SALES'SUMMARIES.
FILE       SALES'SLIP Hollerith 25000 Rigid 72 V(UNREADY) V(READY) V(BUSY)
                    V(ERROR) CARD'READER $
ITEM             CARD Hollerith 72 $            DEFINE        IS '' '' $
DEFINE          CLERK ''BYTE($ø,4$)(CARD)'' $   DEFINE        OF '' '' $
DEFINE     DEPARTMENT ''BYTE($5,4$)(CARD)'' $   DEFINE      PLUS ''+'' $
DEFINE        ARTICLE ''BYTE($1ø,5$)(CARD)'' $  DEFINE     TIMES ''*'' $
DEFINE       QUANTITY ''BYTE($16,4$)(CARD)'' $  DEFINE    EQUALS ''EQ'' $
DEFINE        DOLLARS ''BYTE($22,4$)(CARD)'' $  DEFINE LESS'THAN ''LS'' $
DEFINE          CENTS ''BYTE($27,2$)(CARD)'' $  DEFINE NUMBER'OF ''NENT'' $
TABLE    SALES'CLERKS Variable 5øøø Serial Dense $''Indexed by''
                    BEGIN                          ''clerk number''
ITEM  COMMISSION'RATE fixed 15 Unsigned 15 $
ITEM      COMMISSIONS fixed 15 Unsigned $''In cents''
ITEM     SALES'VOLUME fixed 2ø Unsigned $''In cents''
ITEM     SALES'NUMBER fixed 1ø Unsigned $
                    END
TABLE     DEPARTMENTS Variable 1øøø Serial Dense $''Indexed by''
                    BEGIN                          ''department number''
ITEM  VOLUME'OF'SALES fixed 25 Unsigned $''In cents''
ITEM  NUMBER'OF'SALES fixed 15 Unsigned $
                    END
TABLE        ARTICLES Variable 35øøø Serial Dense $''Indexed by''
                    BEGIN                          ''article number''
ITEM      NUMBER'SOLD fixed 15 Unsigned $
                    END
```

```
STEPØ1. OPEN INPUT SALES'SLIP $
STEPØ2. INPUT SALES'SLIP CARD $
STEPØ3. IF SALES'SLIP EQUALS V(BUSY) $  GOTO STEPØ3 $
STEPØ4. IF SALES'SLIP EQUALS V(ERROR) $  GOTO STEP15 $
STEPØ5. IF SALES'SLIP EQUALS V(READY) $
            BEGIN
STEPØ6. FOR C = NUMBER OF (CLERK) $
        FOR D = NUMBER OF (DEPARTMENT) $
        FOR A = NUMBER OF (ARTICLE) $
                  BEGIN
STEPØ7. IF C IS LESS'THAN NUMBER'OF (SALES'CLERKS) AND D IS LESS'THAN
        NUMBER'OF (DEPARTMENTS) AND A IS LESS'THAN NUMBER'OF (ARTICLES) $
                        BEGIN
STEPØ8. SALES'NUMBER OF ($C$) = SALES'NUMBER OF ($C$) PLUS 1 $
STEPØ9. NUMBER'OF'SALES OF ($D$) = NUMBER'OF'SALES OF ($D$) PLUS 1 $
STEP1Ø. FOR P = 1ØØ TIMES NUMBER OF (DOLLARS) PLUS NUMBER OF (CENTS) $
                              BEGIN
STEP11. SALES'VOLUME OF ($C$) = SALES'VOLUME OF ($C$) PLUS P $
STEP12. VOLUME'OF'SALES OF ($D$) = VOLUME'OF'SALES OF ($D$) PLUS P $
                              END
STEP13. NUMBER'SOLD OF ($A$) = NUMBER'SOLD OF ($A$) PLUS NUMBER OF
        (QUANTITY) $  GOTO STEPØ2 $
             END  END  END
STEP14. FOR C = ALL (SALES'CLERKS) $  COMMISSIONS OF ($C$) =
        COMMISSION'RATE OF ($C$) TIMES SALES'VOLUME OF ($C$) $
STEP15. SHUT INPUT SALES'SLIP $
        END''UPDATE'SALES'SUMMARIES''
```

The flow chart for this statement is given below.



OUTPUT STATEMENTS. A file may be written, one record at a time, by the execution of a sequence of OUTPUT statements. The first statement executed in such a sequence must be an OPEN OUTPUT statement, which activates the file, and the last statement executed must be a SHUT OUTPUT statement, which deactivates the file.

An OPEN OUTPUT statement is composed of the OPEN file operator, followed by the OUTPUT file operator, a file name, an optional constant or specification of a block of values, and terminated by the $ separator.

statement ≠ OPEN OUTPUT name$_{\text{of-inactive-file}}$ ⌊constant;block⌋ $

An OPEN OUTPUT statement activates an inactive file and positions it to zero. If a constant or a block of values is specified, it also initiates a write operation that will transfer the machine symbol representing the constant or specified block of values from the computer's internal memory out to the file as its first record, thus incrementing the file's position by one.

An OUTPUT statement is composed of the OUTPUT file operator followed by a file name, a constant or the specification of a block of values, and terminated by the $ separator.

statement ∉ OUTPUT name$_{\text{of-active-file}}$ constant;block $

An OUTPUT statement initiates a write operation on an inactive file that will transfer the machine symbol representing the constant or the specified block of values from the computer's internal memory out to the file as a record, thus incrementing the file's position by one.

A SHUT OUTPUT statement is composed of the SHUT file operator followed by the OUTPUT file operator, a file name, an optional constant or specification of a block of values, and terminated by the $ separator.

statement ∉ SHUT OUTPUT name$_{\text{of-active-file}}$ ⌊constant;block⌋ $

A SHUT OUTPUT statement deactivates an active file. If a constant or a block of values is specified, it also initiates a final write operation that will, prior to the deactivation of the file, transfer the machine symbol representing the constant or specified block of values from the computer's internal memory out to the file, as its last record.

The following routine, UPDATE'ACCOUNTS, shows how a file may be written by the execution of a sequence of OUTPUT statements. This routine reads a file of checking account records and a file of transaction records. Each account record contains an account number and a balance, and each transaction record contains an account number and either a deposit or a withdrawal, and both files are arranged in order of ascending account number. The routine uses the transaction file to update the account file, writing an updated account file and a file of rejected transactions -- rejected either because they are out of sequence, or because they have left a negative balance.

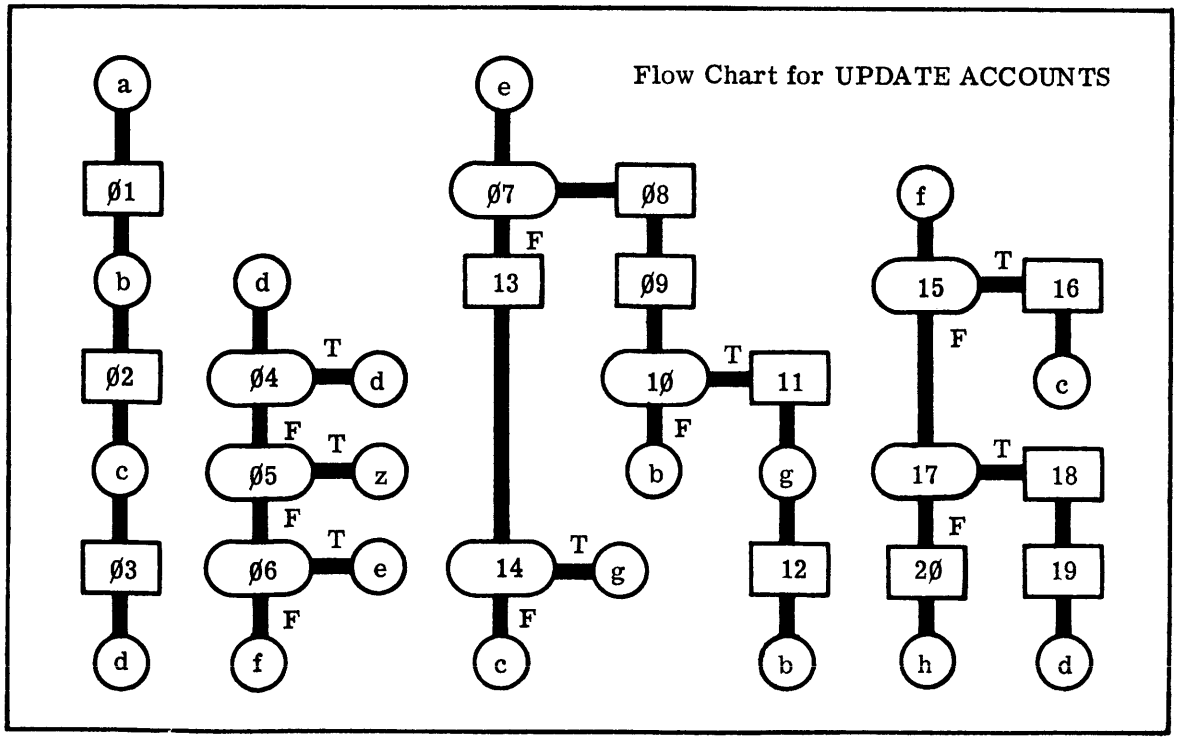
```
                BEGIN UPDATE'ACCOUNTS.
FILE      ACCOUNT Binary 32768 Rigid 36 V(UNREADY) V(READY) V(BUSY)
                V(ERROR) TAPE'A $
FILE TRANSACTION Binary 5ØØØ Rigid 36 V(UNREADY) V(READY) V(BUSY)
                V(ERROR) TAPE'B $
FILE UPDATED'ACCOUNT Binary 32768 Rigid 36 V(UNREADY) V(READY) V(BUSY)
                V(ERROR) TAPE'C $
FILE REJECTED'TRANSACTION Binary 5ØØØ Rigid 36 V(UNREADY) V(READY)
                V(BUSY) V(ERROR) TAPE'D $
TABLE      RECORD Rigid 2 Dense $
                BEGIN
ITEM       NUMBER fixed 15 Unsigned $''Account number''
ITEM      BALANCE fixed 21 Signed $''In cents.  Account file.  Negative
                balance means overdrawn.''
ITEM      DEPOSIT fixed 21 Signed $''In cents.  Transaction file.
                Negative deposit means withdrawal.''
OVERLAY         BALANCE=DEPOSIT $
                END
DEFINE         TO '' '' $
DEFINE       FROM '' '' $

          STEPØ1. OPEN INPUT TRANSACTION $
                  OPEN INPUT ACCOUNT $
                  OPEN OUTPUT REJECTED'TRANSACTION $
                  OPEN OUTPUT UPDATED'ACCOUNT $
          STEPØ2. INPUT TRANSACTION TO RECORD($Ø$) $
          STEPØ3. INPUT ACCOUNT TO RECORD($1$) $
          STEPØ4. IF TRANSACTION EQ V(BUSY) OR ACCOUNT EQ V(BUSY) OR
REJECTED'TRANSACTION EQ V(BUSY) OR UPDATED'ACCOUNT EQ V(BUSY) $
                        GOTO STEPØ4 $
          STEPØ5. IF TRANSACTION EQ V(ERROR) OR ACCOUNT EQ V(ERROR) OR
REJECTED'TRANSACTION NQ V(READY) OR UPDATED'ACCOUNT NQ V(READY) $
                        GOTO ERROR'ROUTINE $
          STEPØ6. IF TRANSACTION EQ V(READY) AND ACCODNT EQ V(READY) $
                        BEGIN
          STEPØ7.       IF NUMBER($Ø$) EQ NUMBER($1$) $
                              BEGIN
          STEPØ8.             BALANCE($1$) = BALANCE($1$)+DEPOSIT($Ø$) $
          STEPØ9.             OUTPUT UPDATED'ACCOUNT FROM RECORD($1$) $
          STEP1Ø.             IF BALANCE($1$) LS Ø $
                                    BEGIN
          STEP11.                   DEPOSIT($Ø$) = Ø $
          STEP12.                   OUTPUT REJECTED'TRANSACTION FROM
RECORD($Ø$) $                       END

                              GOTO STEP Ø2 $
                              END
```

```
STEP13.      OUTPUT UPDATED'ACCOUNT FROM RECORD($1$) $
STEP14.      IF NUMBER($Ø$) LS NUMBER($1$) $
                 GOTO STEP12 $
             GOTO STEPØ3 $
             END
STEP15. IF TRANSACTION EQ V(UNREADY) AND ACCOUNT EQ V(READY) $
             BEGIN
STEP16.      OUTPUT UPDATED'ACCOUNT FROM RECORD($1$) $
             GOTO STEPØ3 $
             END
STEP17. IF TRANSACTION EQ V(READY) AND ACCOUNT EQ V(UNREADY) $
             BEGIN
STEP18.      OUTPUT REJECTED'TRANSACTION FROM RECORD($Ø$) $
STEP19.      INPUT TRANSACTION TO RECORD($Ø$) $
             GOTO STEPØ4 $
             END
STEP2Ø. SHUT INPUT TRANSACTION $
        SHUT INPUT ACCOUNT $
        SHUT OUTPUT UPDATED'ACCOUNT $
        SHUT OUTPUT REJECTED'TRANSACTION $
        END''UPDATE'ACCOUNTS''
```

The sequence of statement executions for UPDATE'ACCOUNTS is graphically illustrated below.



Where the file characteristics allow, a file may be both written and read, one record at a time, by a sequence of INPUT and OUTPUT statements. The first statement executed in such a sequence must be an OPEN statement, which activates the file, and the last statement executed must be a SHUT statement, which deactivates the file. As an example of this, consider the following routine, which computes a solution vector for a system of linear equations* stored on a magnetic drum as a matrix of coefficients.

```
                BEGIN SOLVE'LINEAR'EQUATION'SYSTEM.
DEFINE      RANK ''as the number of equations and unknowns, less than
                 1ØØØ, to be filled in by the routine's user.'' $
DEFINE      SIZE ''as (RANK+1)*RANK, also to be filled in by the
                 routine's user.'' $
DEFINE      WORD ''as the number of bits in a computer word, to be
                 filled in by the routine's user.'' $
FILE COEFFICIENT Binary SIZE Rigid WORD V(UNREADY) V(READY) V(BUSY)
                 V(ERROR) DRUM $
ARRAY   SOLUTION RANK Floating Rounded $
MODE             Floating Rounded $
ARRAY NORMALIZED RANK Boolean $
ARRAY        ROW RANK fixed 1Ø Unsigned $
```

---

* A system of n linear equations in n unknowns

$$\begin{array}{ccccc} a_{\emptyset,\emptyset}x_{\emptyset} & a_{\emptyset,1}x_{1} & \cdots & a_{\emptyset,n-1}x_{n-1} & = a_{\emptyset,n} \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n-1,\emptyset}x_{\emptyset} & a_{n-1,1}x_{1} & \cdots & a_{n-1,n-1}x_{n-1} & = a_{n-1,n} \end{array}$$

can be represented as an n by n+1 matrix of coefficients

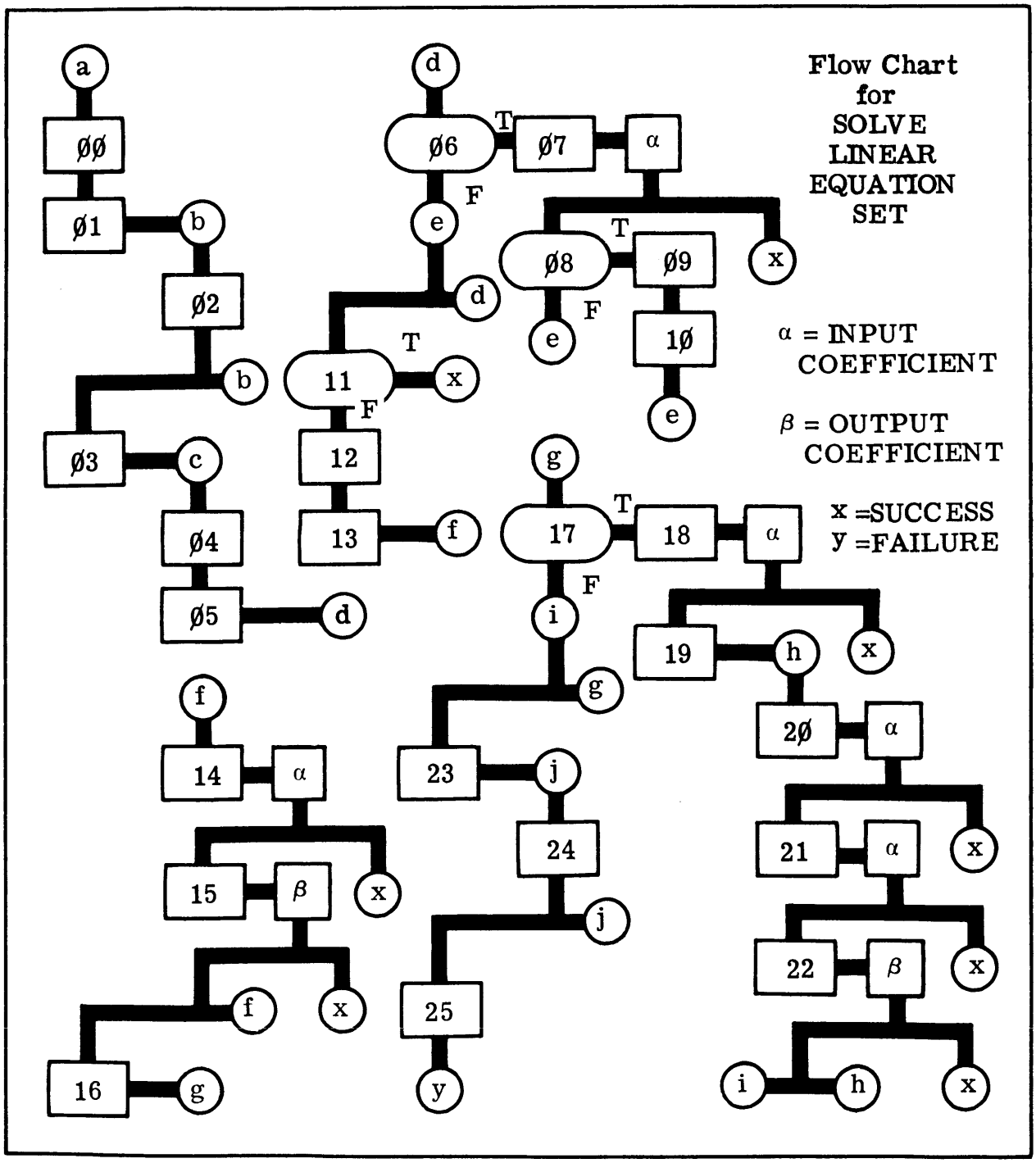
$$\begin{array}{cccc} a_{\emptyset,\emptyset} & a_{\emptyset,1} & \cdots & a_{\emptyset,n-1} & a_{\emptyset,n} \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n-1,\emptyset} & a_{n-1,1} & \cdots & a_{n-1,n-1} & a_{n-1,n} \end{array}$$

For large n, this matrix may exceed available memory space, and must therefore be stored as a file in some external storage device. It may be assumed that record number ((n+1)*i+j) represents $a_{i,j}$, the coefficient in row i and column j of the matrix.

```
PROCedure INPUT'COEFFICIENT (ROW,COLUMN=ELEMENT,ERROR.) $
ITEM      ROW    fixed 1Ø Unsigned $
ITEM      COLUMN fixed 1Ø Unsigned $
                 BEGIN
                 POSition (COEFFICIENT) = (RANK+1)*ROW+COLUMN $
                 INPUT COEFFICIENT ELEMENT $
              XX. IF COEFFICIENT EQ V(BUSY) $  GOTO XX $
                 IF COEFFICIENT EQ V(ERROR) $  GOTO ERROR $
                 END
PROCedure OUTPUT'COEFFICIENT (ROW,COLUMN,ELEMENT=ERROR.) $
ITEM      ROW    fixed 1Ø Unsigned $
ITEM      COLUMN fixed 1Ø Unsigned $
                 BEGIN
                 POSition (COEFFICIENT) = (RANK+1)*ROW+COLUMN $
                 OUTPUT COEFFICIENT ELEMENT $
              XX. IF COEFFICIENT EQ V(BUSY) $  GOTO XX $
                 IF COEFFICIENT EQ V(ERROR) $  GOTO ERROR $
                 END
```

```
STEPØØ. OPEN INPUT COEFFICIENT $
''All rows must initially be unnormalized, steps 1,2.''
STEPØ1. FOR R = Ø,1,RANK-1 $
STEPØ2.      NORMALIZED($R$) = Ø $
''Compute a solution, steps 3...22.''
STEPØ3. FOR C = Ø,1,RANK-1 $
            BEGIN
''Search column C for the row of its highest, unnormalized, non-zero
value, steps 4...11.''
STEPØ4.      HIGH'COLUMN'VALUE = Ø. $
STEPØ5.      FOR R = Ø,1,RANK-1 $
                  BEGIN
STEPØ6.           IF NOT NORMALIZED($R$) $
                       BEGIN
STEPØ7.                INPUT'COEFFICIENT (R,C=ELEMENT,FAILURE.) $
STEPØ8.                IF (/ELEMENT/) GR HIGH'COLUMN'VALUE $
                            BEGIN
STEPØ9.                     HIGH'COLUMN'VALUE = (/ELEMENT/) $
STEP1Ø.                     ROW($C$) = R $
                  END  END  END
STEP11.      IF HIGH'COLUMN'VALUE EQ Ø. $  GOTO FAILURE $
''Normalize this row, steps 12...15.''
STEP12.      NORMALIZED($ROW($C$)$) = 1 $
STEP13.      FOR I = C,1,RANK $
                  BEGIN
STEP14.           INPUT'COEFFICIENT (ROW($C$),I=ELEMENT,FAILURE.) $
STEP15.           OUTPUT(COEFFICIENT (ROW($C$),I,ELEMENT/HIGH'COLUMN'VALUE
                  = FAILURE.) $
                  END
''Clear other coefficients in column C, steps 16...22.''
STEP16.      FOR R = Ø,1,RANK-1 $
                  BEGIN
STEP17.           IF R NQ ROW($C$) $
                       BEGIN
STEP18.                INPUT'COEFFICIENT (R,C=TEMPA,FAILURE.) $
STEP19.                FOR I = C,1,RANK $
                            BEGIN
STEP2Ø.                     INPUT'COEFFICIENT (ROW($C$),I=TEMPB,FAILURE.) $
STEP21.                     INPUT'COEFFICIENT (R,I=ELEMENT,FAILURE.) $
STEP22.                     OUTPUT'COEFFICIENT (R,I,ELEMENT-TEMPA*TEMPB
                            = FAILURE.) $
            END  END  END  END
''Find Solution, steps 23,24.''
STEP23. FOR C = Ø,1,RANK-1 $
STEP24.      INPUT'COEFFICIENT (ROW($C$),RANK=SOLUTION($C$),FAILURE.) $
STEP25. SHUT OUTPUT COEFFICIENT $
        GOTO SUCCESS $
        END''SOLVE'LINEAR'EQUATION'SET''
```

The above routine, though mathematically (and computationally) unsophisticated, could be used as the framework for a more satisfactory algorithm. It serves here to show how a file may be both written and read. The sequence of statement executions for the routine is illustrated in the following diagram.



EXERCISE (Input/Output and Files)

(a) Each record of a warehouse inventory file contains: article name; reorder quantity; quantity on hand; and quantity on back order. Each record of an inventory transaction file contains serial number, quantity, and one of the following transaction status codes: received from supplier; shipped to customer; shipped to customer on back order; returned by customer; returned to supplier; change in reorder quantity. Write a JOVIAL routine to generate an updated inventory file and a file of data on articles where the quantity on hand is below the reorder quantity. Each record of this second file should contain: article name; serial number; quantity on hand - quantity on back order; and reorder quantity. Make any convenient assumptions about the structure of the files involved.

(b) Given the inventory file described above and a file of prices in which each record contains a serial number and a price (in cents). Write a JOVIAL routine to create (1) a priced inventory file in which the item PRICE has been added to each record, and (2) an unpriced inventory file without the item PRICE, for those articles not included in the file of prices.

(c) Given two similarly structured and ordered files, where each record has a 6-character, Hollerith-coded identification on which the records of both files are sequenced. Write a JOVIAL routine to merge these two files into a third, also ordered file. Assume any convenient format for the records of the files.

PROGRAMS

A JOVIAL program is a list of declarations and statements enclosed in the START and TERM brackets. If a statement name is not provided after the TERM, the first statement in the program's execution sequence is the first statement listed that is not part of a procedure declaration. And if this first listed statement is named, its name can also be considered as the name of the program. The $ separator indicates the typographic end of the program.

program ≠ START ⌊declarations⌋ ⌊name$_{\text{of-program}}$.⌋ statements TERM

⌊name$_{\text{of-first-statement-to-be-executed}}$⌋ $

Little can be said about constructing programs that has not already been said about constructing their major components: declarations and statements. It is important to realize, however, that the examples in this manual, though occasionally complex, do not begin to approach the complexity required of actual programs that solve actual problems. The following program, a PAYROLL COMPUTATION, is a good example of this. An actual payroll computation program for a large, multi-structured firm might easily require a report as large as this manual for complete documentation. It would therefore be well to consider the program below in the light of the many simplifying assumptions that have been made. The first of these is the relegation of all the hard and uninteresting parts of the program to a separate existence as COMPOOL procedures. The headings for these procedures are given below, and their parameters should be self-explanatory. They are all functions.

```
PROCedure BINARY'OF                        (HOLLERITH'NUMBER) $
ITEM      BINARY'OF                        fixed 2Ø Unsigned Ø...999999 $
ITEM      HOLLERITH'NUMBER                 Hollerith 6 $

PROCedure HOLLERITH'OF                     (BINARY'NUMBER) $
ITEM      HOLLERITH'OF                     Hollerith 6 $
ITEM      BINARY'NUMBER                    fixed 2Ø Unsigned Ø...999999 $

PROCedure COMPUTED'GROSS'PAY               (MAN'NUMBER,HOURS) $
ITEM      COMPUTED'GROSS'PAY               fixed 17 Unsigned Ø...99999 $
ITEM      MAN'NUMBER                       fixed 13 Unsigned Ø...4999  $
ITEM      HOURS                            fixed Ø7 Unsigned Ø...99    $

PROCedure COMPUTED'FEDERAL'WITHOLDING      (MAN'NUMBER,GROSS'PAY) $
ITEM      COMPUTED'FEDERAL'WITHOLDING      fixed 17 Unsigned Ø...99999 $
ITEM      MAN'NUMBER                       fixed 13 Unsigned Ø...4999  $
ITEM      GROSS'PAY                        fixed 17 Unsigned Ø...99999 $

PROCedure COMPUTED'FICA                    (MAN'NUMBER,GROSS'PAY) $
ITEM      COMPUTED'FICA                    fixed 14 Unsigned Ø...9999  $
ITEM      MAN'NUMBER                       fixed 13 Unsigned Ø...4999  $
ITEM      GROSS'PAY                        fixed 17 Unsigned Ø...99999 $

PROCedure COMPUTED'STATE'WITHOLDING        (MAN'NUMBER,GROSS'PAY) $
ITEM      COMPUTED'STATE'WITHOLDING        fixed 14 Unsigned Ø...9999  $
ITEM      MAN'NUMBER                       fixed 13 Unsigned Ø...4999  $
ITEM      GROSS'PAY                        fixed 17 Unsigned Ø...99999 $

PROCedure COMPUTED'RETIREMENT              (MAN'NUMBER,ADJUSTED'GROSS) $
ITEM      COMPUTED'RETIREMENT              fixed 14 Unsigned Ø...9999  $
ITEM      MAN'NUMBER                       fixed 13 Unsigned Ø...4999  $
ITEM      ADJUSTED'GROSS                   fixed 17 Unsigned Ø...99999 $
```

```
PROCedure COMPUTED'MEDICAL'PLAN      (MAN'NUMBER,ADJUSTED'GROSS) $
ITEM      COMPUTED'MEDICAL'PLAN      fixed 14 Unsigned ø...9999  $
ITEM      MAN'NUMBER                 fixed 13 Unsigned ø...4999  $
ITEM      ADJUSTED'GROSS             fixed 17 Unsigned ø...99999 $

PROCedure COMPUTED'MISCELLANEOUS     (MAN'NUMBER,ADJUSTED'GROSS) $
ITEM      COMPUTED'MISCELLANEOUS     fixed 14 Unsigned ø...9999  $
ITEM      MAN'NUMBER                 fixed 13 Unsigned ø...4999  $
ITEM      ADJUSTED'GROSS             fixed 17 Unsigned ø...99999 $
```

In addition to their parameters, most of these COMPOOL procedures use data from a file of PERSONNEL RECORDS. The PERSONNEL file and the RECORDS table that describes its format are also, conveniently, declared in the COMPOOL. These declarations are partly given below; the body of the table declaration has been omitted.

```
FILE      PERSONNEL Binary 1 Variable 15øøøøø V(UNREADY) V(READY)
               V(BUSY) V(ERROR) TAPE'A $
TABLE     RECORDS Variable 5øøø Serial Dense $
```

The PAYROLL COMPUTATION program, which automatically incorporates the COMPOOL declarations incompletely given above, reads in a deck of employee time cards and writes, on the line printer, paychecks for these employees. The employee's time cards are punched with the employee's name, his number, the hours he's worked, and his department code, as shown below.

```
A.B. WORKER          3333  4ø  XYZ
```

The deck of time cards must be prefaced with a card giving the date, as follows:

```
CURRENT DATE IS     13 OCT 61
```

The paychecks that are the program's output are printed on continuous paycheck forms, with four lines of print per paycheck. Paycheck format is defined in the program, but a sample paycheck printout is shown below.

```
3333   XYZ   13 OCT 61   13784     8444
267ø    74ø  11øø     33ø   5øø          534ø
     A.B. WORKER                          84.44
***EIGHTY FOUR DOLLARS AND FORTY FOUR CENTS***
```

The program also updates certain accumulated totals in the PERSONNEL RECORDS file, such as year-to-date gross earnings, etc., but as this is done by the appropriate COMPOOL procedure, (e.g., COMPUTED'GROSS'PAY), it need not concern us here.

The operating procedures for the PAYROLL COMPUTATION program are:

1. Load program into memory.

2. Ready inputs and outputs.

    a. Place Date Card followed by deck of Employee Time Cards in card reader.

    b. Insert continuous Paycheck form in line printer.

    c. Place tape containing Personnel file on tape drive "A".

3. Execute the program.

4. Program stops:

    a. If the program stops at step 11, then ready card reader and continue program execution.

    b. If the program stops at step 54, ready line printer and continue program execution.

    c. If the program stops at step 6Ø, either it has completed its operation or it hasn't started. If no checks have been printed, begin over.

```
          START PAYROLL'COMPUTATION.
          BEGIN
FILE      TIME Hollerith 5ØØØ Rigid 72 V(UNREADY) V(READY) V(BUSY)
               V(ERROR) CARD'READER $
ITEM      CARD Hollerith 72 $
               ''Date Card Format''
DEFINE    CURRENT'DAY          ''BYTE($2Ø,2$)(CARD)'' $
DEFINE    CURRENT'MONTH        ''BYTE($23,3$)(CARD)'' $
DEFINE    CURRENT'YEAR         ''BYTE($27,2$)(CARD)'' $
               ''Employee Card Format''
DEFINE    EMPLOYEE'NAME        ''BYTE($Ø,2Ø$)(CARD)'' $
DEFINE    EMPLOYEE'NUMBER      ''BYTE($22,4$)(CARD)'' $
DEFINE    HOURS'WORKED         ''BYTE($28,2$)(CARD)'' $
DEFINE    DEPARTMENT'CODE      ''BYTE($32,3$)(CARD)'' $

FILE      PAY Hollerith 5ØØØ Rigid 72 V(UNREADY) V(READY) V(BUSY)
               V(ERROR) LINE'PRINTER $
ITEM      CHECK Hollerith 72 $
               ''Check Format, Line Ø''
DEFINE    NUMBER               ''BYTE($ØØ,4$)(CHECK)'' $
DEFINE    DEPARTMENT           ''BYTE($Ø7,3$)(CHECK)'' $
DEFINE    DAY                  ''BYTE($13,2$)(CHECK)'' $
DEFINE    MONTH                ''BYTE($16,3$)(CHECK)'' $
DEFINE    YEAR                 ''BYTE($2Ø,2$)(CHECK)'' $
DEFINE    GROSS                ''BYTE($25,5$)(CHECK)'' $
DEFINE    NET                  ''BYTE($34,5$)(CHECK)'' $
DEFINE    NET'DOLLARS          ''BYTE($34,3$)(CHECK)'' $
DEFINE    NET'CENTS            ''BYTE($37,2$)(CHECK)'' $
               ''Check Format, Line 1''
DEFINE    FEDERAL'WITHOLDING   ''BYTE($ØØ,5$)(CHECK)'' $
DEFINE    STATE'WITHOLDING     ''BYTE($Ø7,4$)(CHECK)'' $
DEFINE    FICA                 ''BYTE($13,4$)(CHECK)'' $
DEFINE    RETIREMENT           ''BYTE($21,4$)(CHECK)'' $
DEFINE    MEDICAL'PLAN         ''BYTE($27,4$)(CHECK)'' $
DEFINE    MISCELLANEOUS        ''BYTE($33,4$)(CHECK)'' $
DEFINE    TOTAL'DEDUCTIONS     ''BYTE($4Ø,5$)(CHECK)'' $
               ''Check Format, Line 2''
DEFINE    NAME                 ''BYTE($5,2Ø$)(CHECK)'' $
DEFINE    AMOUNT'DOLLARS       ''BYTE($44,3$)(CHECK)'' $
DEFINE    DECIMAL'POINT        ''BYTE($47$)(CHECK)'' $
DEFINE    AMOUNT'CENTS         ''BYTE($48,2$)(CHECK)'' $
               ''Check Format, Line 3''
DEFINE    ENGLISH'AMOUNT       ''CHECK'' $

SWITCH    FILL'IN'LINE = (LINEØ,LINE1,LINE2,LINE3) $
```

```
PROCedure NUMERIC'TO'ENGLISH'CONVERSION (NUMERIC'VALUE=SIZE,
          ENGLISH'EQUIVALENT) $
ITEM      NUMERIC'VALUE fixed 1Ø Unsigned Ø...999 $
ITEM      SIZE fixed 5 Unsigned Ø...3Ø $
ITEM      ENGLISH'EQUIVALENT Hollerith 3Ø $
TABLE     NUMERIC'TO'ENGLISH Rigid 9 $
               BEGIN
ITEM      UNIT Hollerith 5 $  BEGIN 5H(ONE  ) 5H(TWO  ) 5H(THREE)
               5H(FOUR ) 5H(FIVE ) 5H(SIX  ) 5H(SEVEN) 5H(EIGHT)
               5H(NINE ) END
ITEM      UNIT'SIZE fixed 3 Unsigned $  BEGIN 3 3 5 4 4 3 5 5 4 END
ITEM      TEEN Hollerith 9 $  BEGIN 9H(ELEVEN   ) 9H(TWELVE   )
               9H(THIRTEEN ) 9H(FOURTEEN ) 9H(FIFTEEN  ) 9H(SIXTEEN  )
               9H(SEVENTEEN) 9H(EIGHTEEN ) 9H(NINETEEN ) END
ITEM      TEEN'SIZE fixed 4 Unsigned $  BEGIN 6 6 8 8 7 7 9 8 8 END
ITEM      TEN Hollerith 7 $  BEGIN 7H(TEN    ) 7H(TWENTY ) 7H(THIRTY )
               7H(FORTY  ) 7H(FIFTY  ) 7H(SIXTY  ) 7H(SEVENTY)
               7H(EIGHTY ) 7H(NINETY ) END
ITEM      TEN'SIZE fixed 3 Unsigned $  BEGIN 3 6 6 6 5 5 7 6 6 END
               END
          BEGIN
          ENGLISH'EQUIVALENT = 3ØH(                                  ) $
          SIZE = Ø $
          FOR A = Ø $
          FOR B = Ø $
          FOR C = Ø $
               BEGIN
               REMQUO (NUMERIC'VALUE,1ØØ=A,B) $
               REMQUO (B,1Ø=B,C) $
               IF A NQ Ø $
                    BEGIN
                    BYTE($Ø,5$)(ENGLISH'EQUIVALENT) = UNIT($A-1$) $
                    SIZE = UNIT'SIZE($A-1$)+8 $
                    BYTE($SIZE-7,7$)(ENGLISH'EQUIVALENT) = 7H(HUNDRED) $
                    END
               IF B EQ 1 AND C NQ Ø $
                    BEGIN
                    BYTE($SIZE+1,9$)(ENGLISH'EQUIVALENT) = TEEN($C-1$) $
                    SIZE = SIZE+TEEN'SIZE($C-1$)+1 $
                    RETURN $
                    END
               IF B NQ Ø $
                    BEGIN
                    BYTE($SIZE+1,7$)(ENGLISH'EQUIVALENT) = TEN($B-1$) $
                    SIZE = SIZE+TEN'SIZE($B-1$)+1 $
                    END
               IF C NQ Ø $
                    BEGIN
                    BYTE($SIZE+1,5$)(ENGLISH'EQUIVALENT) = UNIT($C-1$) $
                    SIZE = SIZE+UNIT'SIZE($C-1$)+1 $
          END  END  END
```

```
STEPØ1.   OPEN INPUT PERSONNEL RECORDS $
STEPØ2.   OPEN INPUT TIME CARD $
STEPØ3.   OPEN OUTPUT PAY $
STEPØ4.   IF PERSONNEL EQ V(BUSY) OR TIME EQ V(BUSY) $  GOTO STEPØ4 $
STEPØ5.   IF PERSONNEL EQ V(READY) AND TIME EQ V(READY) $
               BEGIN
STEPØ6.        DAY = CURRENT'DAY $
STEPØ7.        MONTH = CURRENT'MONTH $
STEPØ8.        YEAR = CURRENT'YEAR $
STEPØ9.        INPUT TIME CARD $
STEPlØ.        IF TIME EQ V(BUSY) $  GOTO STEPlØ $
STEPll.        IF TIME EQ V(ERROR) $  STOP STEPØ9 $
STEPl2.        IF TIME EQ V(READY) $
                    BEGIN
STEPl3.''C       '' FOR A = BINARY'OF (EMPLOYEE'NUMBER) $
STEPl4.'' O      '' FOR B = BINARY'OF (HOURS'WORKED) $
STEPl5.''  M     '' FOR C = COMPUTED'GROSS'PAY (A,B) $
STEPl6.''   P    '' FOR D = COMPUTED'FEDERAL'WITHOLDING (A,C) $
STEPl7.''P   U   '' FOR E = COMPUTED'STATE'WITHOLDING (A,C) $
STEPl8.'' A   T  '' FOR F = COMPUTED'FICA (A,C) $
STEPl9.''  Y   E '' FOR G = COMPUTED'RETIREMENT (A,C-(D+E+F)) $
STEP2Ø.''   C    '' FOR H = COMPUTED'MEDICAL'PLAN (A,C-(D+E+F+G)) $
STEP2l.''D   H   '' FOR I = COMPUTED'MISCELLANEOUS (A,C-(D+E+F+G+H)) $
STEP22.'' A   E  '' FOR J = D+E+F+G+G+I $
STEP23.''  T   C '' FOR K = C-J $
STEP24.''   A   K'' FOR L = Ø,1,3 $
                         BEGIN
STEP25.                  CHECK = 1H( ) $
STEP26.                  GOTO FILL'IN'LINE ($L$) $
STEP27.            LINEØ. NUMBER = EMPLOYEE'NUMBER $
STEP28.                  DEPARTMENT = DEPARTMENT'CODE $
STEP29.                  GROSS = HOLLERITH'OF (C) $
STEP30.                  NET = HOLLERITH'OF (K) $
                         GOTO STEP52 $
STEP31.            LINEl. FEDERAL'WITHOLDING = HOLLERITH'OF (D) $
STEP32.                  STATE'WITHOLDING = HOLLERITH'OF (E) $
STEP33.                  FICA = HOLLERITH'OF (F) $
STEP34.                  RETIREMENT = HOLLERITH'OF (G) $
STEP35.                  MEDICAL'PLAN = HOLLERITH'OF (H) $
STEP36.                  MISCELLANEOUS = HOLLERITH'OF (I) $
STEP37.                  TOTAL'DEDUCTIONS = HOLLERITH'OF (J) $
STEP38.            LINE2. NAME = EMPLOYEE'NAME $
STEP39.                  AMOUNT'DOLLARS = NET'DOLLARS $
STEP4Ø.                  DECIMAL'POINT = 1H(.) $
STEP41.                  AMOUNT'CENTS = NET'CENTS $
                         GOTO STEP52 $
```

```
STEP42.          LINE3. ENGLISH'AMOUNT
                        = 3ØH(***                              ) $
STEP43.                 FOR X = 3 $
STEP44.                 FOR Y = Ø $
STEP45.                 FOR Z = BINARY'OF (AMOUNT'DOLLARS) $
                              BEGIN
STEP46.                       GOTO CONVERT $
STEP47.                       BYTE($X,13$)(ENGLISH'AMOUNT)
                              = 13H( DOLLARS AND ) $
STEP48.                       X = X+13 $
STEP49.                       Z = BINARY'OF (AMOUNT'CENTS) $
STEP5Ø.                       GOTO CONVERT $
STEP51.                       BYTE($X,9$)(ENGLISH'AMOUNT)
                              = 9H( CENTS***) $
                              GOTO STEP52 $
                              CLOSE
                    CONVERT $       BEGIN
                                    NUMERIC'TO'ENGLISH'CONVERSION
                                    (Z=Y,BYTE($X,3Ø$)(ENGLISH'AMOUNT)) $
                                    IF Y EQ Ø $
                                         BEGIN
                                         BYTE($X,2$)(ENGLISH'AMOUNT)
                                         = 2H(NO) $
                                         Y = 2 $
                                         END
                                    X = X+Y $
                                END  END
STEP52.                       OUTPUT PAY CHECK ''LINE($L$)'' $
STEP53.                       IF PAY EQ V(BUSY) $  GOTO STEP53 $
STEP54.                       IF PAY NQ V(READY) $  STOP STEP52 $
                              END
STEP55.                   POSition (PAY) = POSition (PAY) + 2 $
                          GOTO STEPØ9 $
                          END
STEP56.              POSition (PERSONNEL) = Ø $
STEP57.              SHUT OUTPUT PERSONNEL RECORDS $
                     END
STEP58.         SHUT INPUT TIME $
STEP59.         SHUT OUTPUT PAY $
STEP6Ø.         STOP STEPØ1 $
                END
                TERM PAYROLL'COMPUTATION $
```

EXERCISE (Programs)

(a) The SWAC is a small, medium-speed binary computer with but 256 words of high-speed memory. It has a command structure of just 13 instructions, which is, however, quite powerful because the SWAC is a 4-address computer. SWAC interprets its 36-bit memory words in one of two ways: as numbers (sign bit and 35 fractional magnitude bits); and as instructions (4 addresses of 8 bits each, and a 4-bit operation code). The cells of an instruction word are called ALPHA, BETA, GAMMA, DELTA, and OP, where OP is the operation code. At present, writing programs for the SWAC is a rather tedious exercise in punching binary cards; a symbolic assembly program is needed. To be useful, a SWAC assembler should accomplish at least three things:

(1) Translate mnemonic operation codes to their numeric equivalents.

The SWAC Instruction List

MNEMONIC OPERATION CODE	NUMERIC OPERATION CODE	TITLE
INIPUT	ØØ	INItial inPUT
SINPUT	Ø1	Standard INPUT
OUTPUT	Ø2	OUTPUT
BIOADD	Ø4	Branch If Overflow ADDition
NORADD	Ø5	NORmal ADDition
BIOSUB	Ø6	Branch If Overflow SUBtraction
NORSUB	Ø7	NORmal SUBtraction
COMPAR	Ø8	COMPARe
COMMAG	Ø9	COMpare MAGnitudes
SPEMUL	1Ø	SPEcial MULtiplication
NORMUL	11	NORmal MULtiplication
MULTIP	12	double-precision MULTIPlication
ETRASH	14	ExTRAct and SHift

(2) Accept symbolic location labels from a location label field, assign them numeric addresses, and decode them when they occur in an address field. Location labels are left justified and consist of two to six characters: a letter followed by one to five letters or numerals.

(3) Convert octal numbers appearing in an address field or the operation code field. In an address field, an octal number may range from Ø to 377, and in the operation code field, an octal number may range from Ø to 17, right justified in both cases.

The input for SWACAP, the SWAC Assembly Program, is a deck of punched cards, with six fields: Label; Alpha; Beta; Gamma; Delta; Op-code; and Remarks. A blank field indicates a corresponding cell of zeros, and the first card in the deck is to be associated with location zero in the SWAC memory.

The output of SWACAP is an internal table, declared as follows:

```
TABLE MEMORY Rigid 256 Serial Dense $''Memory image of SWAC machine
             language program''
             BEGIN
ITEM    ALPHA fixed  8 Unsigned $
ITEM     BETA fixed  8 Unsigned $
ITEM    GAMMA fixed  8 Unsigned $
ITEM    DELTA fixed  8 Unsigned $
ITEM       OP Status 4 V(INIPUT) V(SINPUT) V(OUTPUT) V(NULLØ) V(BIOADD)
              V(NORADD) V(BIOSUB) V(NORSUB) V(COMPAR) V(COMMAG) V(SPEMUL)
              V(NORMUL) V(MULTIP) V(NULL1) V(ETRASH) V(NULL2) $
              END
```

Write SWACAP in JOVIAL, assuming any convenient card format. Declare whatever environmental elements are required. No indication need be given if a field in an input symbolic instruction contains an error; the corresponding cell in the output machine instruction may be assigned an arbitrary value. Do not, however, expect that no errors will occur.

NOTE: To increase the difficulty and scope of this exercise, many refinements to SWACAP can be specified, for example:

- Error indications;
- The ability to convert an octal constant for an entire word, instead of just four or eight bit chunks. (This could be accomplished by using an extra field, and a pseudo-instruction named OCTNUM.)
- The ability to specify an arbitrary program origin (with the pseudo-instruction ORIGIN and an octal constant in the Alpha field).
- The ability to repeat the next (Alpha) symbolic instructions (Beta) times -- with octal constants in the Alpha and Beta fields, and the pseudo-instruction REPEAT.
- The ability to use a location label to denote a pre-set parameter, with the pseudo-instruction EQUALS and an octal constant in the Alpha field.

(b) The previous exercise involved writing SWACAP, an assembly program for the SWAC, whose output was the 256-entry MEMORY table, an image of SWAC's 256-word memory. In order to code-check SWAC programs thus assembled, an interpreter program, SWACIN, is required. SWACIN

must simulate not only the operation of SWAC and its memory, but also its auxiliary storage and its input/output components.

SWAC's auxiliary storage consists of an 8192-word magnetic drum, which contains 256 channels of 32 words each. Its input/output components consist of this drum, a card reader, a card punch, and a line printer. Input/output transfers in the SWAC are block transfers: an entire 32-word channel for the drum, and 24-word card images for the reader, punch, and printer.

The environment for the SWAC interpreter program, SWACIN, will consist of: the MEMORY table simulating the SWAC's 256-word high-speed memory, as declared in the previous exercise; a DRUM array simulating the SWAC's 8192-word magnetic drum, declared as follows:

```
ARRAY DRUM 256''channels''32''words per channel''fixed 36 Signed 35 $
```

and three binary files, READER, PUNCH, and PRINTER simulating the SWAC's card reader, card punch, and line printer. Each of the rigid length records for these files contains twenty-four 36-bit words, comprising a 72-column, binary card image.

Operational Description of SWAC Instructions

Unless otherwise stated, or a branch occurs, instructions are executed in sequential order, starting with location zero.

```
BIOADD  Branch If Overflow ADDition
        ALPHA:  address of augend
        BETA:   address of addend
        GAMMA:  address of sum
        DELTA:  address of next instruction if overflow

NORADD  NORmal ADDition
        ALPHA:  address of augend
        BETA:   address of addend
        GAMMA:  address of sum
        DELTA:  address of next instruction

BIOSUB  Branch If Overflow SUBtraction
        ALPHA:  address of minuend
        BETA:   address of subtrahend
        GAMMA:  address of difference
        DELTA:  address of next instruction if overflow
```

NORSUB NORmal SUBtraction
- ALPHA: address of minuend
- BETA: address of subtrahend
- GAMMA: address of difference
- DELTA: address of next instruction

SPEMUL SPEcial MULtiplication
- ALPHA: address of multiplier
- BETA: address of multiplicand
- GAMMA: address of most significant part of product, rounded
- DELTA: not used

NORMUL NORmal MULtiplication
- ALPHA: address of multiplier
- BETA: address of multiplicand
- GAMMA: address of most significant part of product, rounded
- DELTA: address of next instruction

MULTIP MULTIPlication, double-precision
- ALPHA: address of multiplier
- BETA: address of multiplicand
- GAMMA: address of most significant part of product
- DELTA: address of least significant part of product

COMPAR COMPARe
- ALPHA: address of minuend
- BETA: address of subtrahend
- GAMMA: address of difference
- DELTA: address of next instruction if difference GQ zero

COMMAG COMpare MAGnitudes
- ALPHA: address of minuend
- BETA: address of subtrahend
- GAMMA: address of difference of magnitudes
- DELTA: address of next instruction if difference GQ zero

ETRASH ExTRAct and SHift
- ALPHA: address of extractor (mask determining bits to be extracted
- BETA: address of extractee
- GAMMA: address of extracted and shifted result (bits of the extractee corresponding to 1 bits of the extractor are set to zero, then shifting occurs)
- DELTA: amount and direction of shift (first bit not used. If second bit is: ϕ -- shift left; 1 -- shift right. Remaining bits specify number of bit positions to shift.

INIPUT INItial inPUT — Input is placed in 24 or 32-word memory block beginning with the register containing this instruction.
- ALPHA: not used
- BETA: not used
- GAMMA: drum channel address, if pertinent; otherwise not used
- DELTA: specifies input device

SINPUT Standard INPUT
- ALPHA: address of first word of memory block for input
- BETA: not used
- GAMMA: drum channel address, if pertinent; otherwise not used
- DELTA: specifies input device

OUTPUT OUTPUT
- ALPHA: address of first word of memory block to output
- BETA: not used
- GAMMA: drum channel address, if pertinent, otherwise not used
- DELTA: specifies output device

For the three input/output instructions, the input or output device is specified by DELTA with the following 3-digit octal code:

Line Printer	ØØØ	(24-word block transfer)
Card Reader	1ØØ	(24-word block transfer)
Card Punch	12Ø	(24-word block transfer)
Drum	16Ø	(32-word block transfer)

All three instructions transfer program control to the next instruction in sequence only after the input/output transfer has been completed. Thus, specifying the printer or punch for input, or the reader for output, will cause the SWAC to halt its operation. An illegal code in the DELTA field will also cause a halt. The most commonly used "stop" instruction, however, is a word of all zero bits.

The spare operation codes, denoted by the statuses NULLØ, NULL1, and NULL2 in the declaration of the OP item, are currently executed by the SWAC as OUTPUT, MULTIP, and ETRASH, respectively. A programming convention, however, forbids such use to avoid program obsolescence when and if new instructions are incorporated into the circuitry of the computer. SWACIN should therefore interpret these instructions as halts.

Write SWACIN in JOVIAL, declaring the three binary files READER, PUNCH, and PRINTER, along with whatever other environmental elements may be required. Where the preceding description of the SWAC is incomplete or ambiguous -- and several important characteristics must be inferred since they are never explicitly stated -- individual judgment must be exercised.

NOTE: Some slight simplifications have been made in the above description of the SWAC computer at U.C.L.A. Programs written for SWACIN would, however, operate perfectly well on the SWAC; even, perhaps, with the same results.

(c) Write, in JOVIAL, a program that will accept and interpretively execute programs coded in SIMPAL (SIMPle Algebraic Language), which is constructed in the following way:

1. number ≠ ⌈Ø;1;2;3;4;5;6;7;8;9⌉s

Numbers are no more than ten digits in length, and thus range in value from Ø to 9999999999.

2. variable ≠ V(:formula:)

There are 1ØØØØ integer valued variables, V(Ø) thru V(9999), which are distinguished from each other by the value of the parenthesized formula. Variables have the same range of values as do numbers.

3. formula ≠ number;variable;⌈(formula)⌉;⌈⌊-⌋ formulas ⌈+;-;*;/;**⌉⌉

Formulas follow the rules of integer arithmetic, where any fractional part of the result of an operation is truncated. They have the same range of values as do variables and numbers.

4. statement ≠ number:) statement

Statements may be labeled by arbitrary numbers.

5. statement ≠ GO number .

The next statement to be executed is the one labeled with the given number.

6. statement ≠ IF formula = formula ⌊... formula⌋ , statement

If the value specified by the formula on the left of the = separator equals the value specified by the formula on the right (or any value in the range of values if one is specified) then the component statement is to be executed; otherwise, it is to be skipped.

7. statement ≠ variable = formula .

The variable is assigned the value specified by the formula.

8. program ≠ START statements FINISH

A SIMPAL program is a string of statements enclosed in the START and FINISH brackets. The statements are normally executed in the sequence in which they are listed.

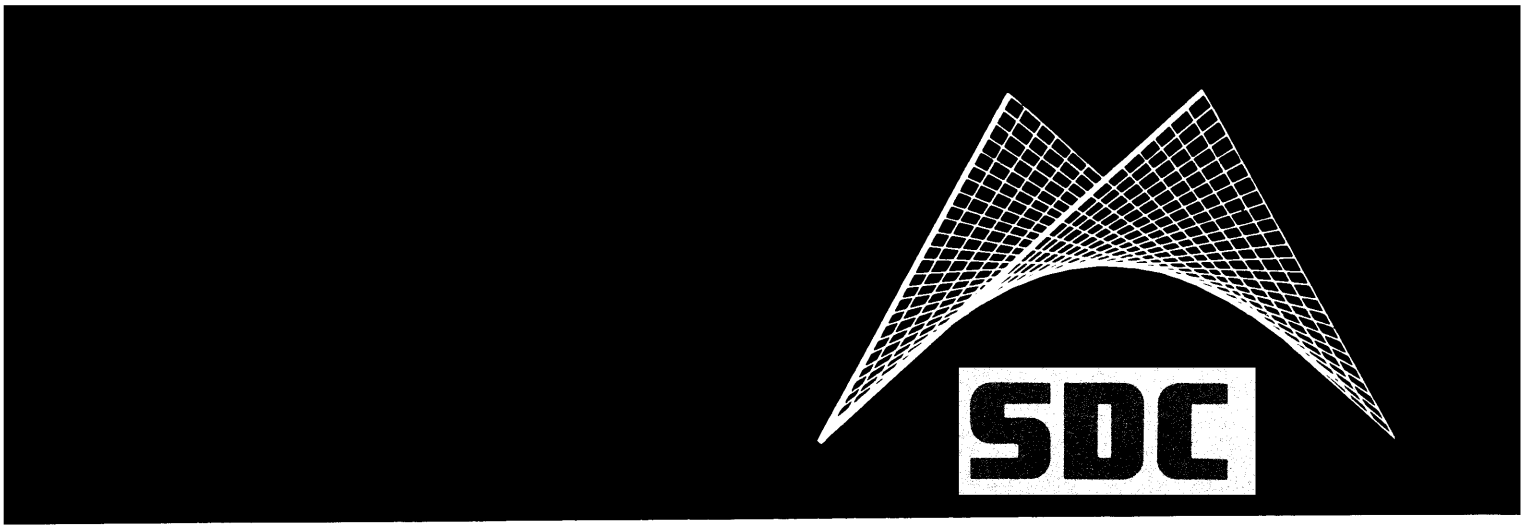
SIMPAL may be illustrated by the accompanying routine, which sets V(Ø) to the sum of V(1ØØ) thru V(199).

```
    V(Ø) = Ø.
    V(1) = Ø.
1)  V(Ø) = V(Ø)+V(1ØØ+V(1)).
    V(1) = V(1)+1.
    IF V(1) = Ø...99, GO 1.
```

26 December 1961 TM-555/003/00

Distribution List:

Name	Location	Name	Location
Anderson, Ruth	9107	Jacoby, Len	Falls Church
Bartram, Phil	9716	Jerew, Dick	4714
Beasley, John	Falls Church	Kneemeyer, John	2416
Beeler, Dick	2421	Koory, Jerry	Falls Church
Blauer, Marty	Paramus	Lewis, Hugh	2051
Book, Erwin	2132	Madden, Don	2163
Bosak, Bob	2217A	Manelowitz, Howard	2129
Bowman, Sally	4070B	Marzocco, Frank	9719A
Bratman, Harvey	2124	McGill, Bob	4070A
Bryans, John	3761	McIsaac, Paul	Lexington
Buckley, Pat	2129	Mosmann, Chuck	9013
Cartmell, Dave	2412	Neeb, Donna	Paramus
Clark, Ellen	9111	Olsen, Ray	2019A
Cohen, Virginia	2414	Ottina, John	3466
Davis, Roger	4706	Perstein, Miff	9106
Deason, Jim	Falls Church	Peterka, Jerry	20130A
Dennis, Weldon	Falls Church	Pulliam, Marv	3462
Dobrusky, Bill	2135	Rafferty, John	2417
Dobbs, Guy	24048	Reiff, Art	2065
Drutz, Bud	Falls Church	Richmond, Hal	Paramus
Englund, Don	2127	Schwartz, Jules	2130
Foote, Ed	20125A	Shasberger, Bill	4470B
Gardner, Stan	4714	Shaw, Chris	2425
Gicie, Pat	2427	Sheffield, Earl	20052
Griffitts, Vance	9109	Shirley, Lynn	2419
Haueisen, Bill	4711	Simmons, Bob	9636
Hayes, Emanuel	9112	Tischhauser, Dick	4062B
Hotelling, George	4711	Weaver, Pat	Paramus
Howell, Hank	2411	Williams, Carle	Lexington
Isbitz, Harold	2415	Winter, Jack	Falls Church
Jackson, Cal	4713		



SYSTEM DEVELOPMENT CORPORATION / 2500 COLORADO AVENUE / SANTA MONICA, CALIFORNIA