SDC FIELD NOTE A WORKING PAPER FN-LO-34-2

AUTHOR
[illegible]
Jules I. Schwartz

APPROVED
Martin Blauer
M. Blauer

PAGE 1 OF PAGES
70

1 February 1960

(Produced under System Development Corporation sub-contract No. 202 issued by International Electric Corporation in performance of contract AF-30(635)-11583.)

JOVIAL - A DESCRIPTION OF THE LANGUAGE

This Document Supersedes

FN-LO-34-1

JOVIAL - REPORT #2

25 May 1959

and

All Supplements

TABLE OF CONTENTS

Cont.

TABLE OF CONTENTS

1. INTRODUCTION

This document contains a complete description of the JOVIAL Language as of February 1, 1960. It reflects the capability of the JOVIAL Interpreter Program which has been delivered as of the same date. As time goes on, additions will be made to the language. It is fairly certain that no addition will obsolete any programming which follows the rules discussed in this document. Changes will be the type which add new areas to the language or remove restrictions which now exist. In any case, these will be noted in future documentation.

1.1 Remarks on Symbology

The symbology used in this document is the hardware language. This is the language which must be used for keypunching at the present time. In many cases the symbols used are the natural symbols for the particular operator or bracket. In other cases, the symbology is rather artificial. These latter symbols will be replaced as soon as hardware is available which permits the punching of the true characters. The following is a list of the less meaningful hardware symbols and their respective "true" meanings.

Hardware Language	True Symbol
$	;
($	[
$)	]
(*	↑
*)	↓

1.2 Remarks on the Contents of this Document

People who are familiar with JOVIAL should have little trouble reading and understanding this document. At most, it might answer a few questions which they have about hitherto unmentioned details. Much of the detail is based on problems and questions which have arisen through use of the language in the past year.

To those who are relatively unfamiliar with the language, it is hoped that the mass of verbiage does not create the illusion of a language which requires many months of intensive study before a coding sheet and pencil may be used.

It is suggested that upon first reading one should attempt to get a general impression rather than try to absorb all of the details and rules (and, on occasion, exceptions to the rules). Chapter 18 has been included as an aid to those striving for a general, rather than a detailed, impression. Once a general impression is obtained, an attempt to use the language for specific problems is recommended. This will usually require references to this document as unknown areas are encountered. From this a gradual feeling of familiarity should occur. A second or third reading after the problem solving phase should serve to make one feel thoroughly acquainted with the language. Of course those who have at their disposal the use of the JOVIAL Interpretor and/or Translator will be aided greatly in their attempt at mastering JOVIAL.

For those who are privileged to attend classes in this language the use of this document will probably be for reference only. Experience has shown that verbal instruction as a supplement to documentation has been extremely beneficial.

2. VARIABLE DEFINITION

One of the striking differences in programming in this language as opposed to programming in machine language is the fact that variables must be defined explicitly. When one wishes to use a register for a temporary computation in machine language, he need only give a location for this value. In the case of JOVIAL, however, one cannot use a variable without having all its characteristics described by the program or the Communication Pool.

Within the program, variables may be described by Variable Declaration Statements. Both tables and items may be described within the program.

There are two uses for defining variables within a program. One is the definition of variables which will be used solely within the program (i.e., not communication items). The other use is to override the Communication Pool when changes are necessary but not yet physically implemented in the system.

2.1 Item Declaration Statements

The format for an item definition depends on the type of coding the item has, although all Item Declaration statements have several fields in common (e.g., the word "ITEM", the name of the item).

Formats for the Item Declaration Statements are as follows:

```
ITEM NAME F$
ITEM NAME I #BITS S/U$
ITEM NAME H #CHARACTERS$
ITEM NAME A #BITS S/U # BITS RIGHT OF BINARY POINT$
ITEM NAME S ST1 ST2 ST2 ...STN$
```

The above symbology has the following meaning:

<u>ITEM</u> declares this to be an Item Declaration Statement.

<u>NAME</u> is the field which contains the name of the item. It <u>is two to six alphanumeric symbols with the first character always a letter</u>.

<u>F</u> is a floating type item. (e.g., full word, characteristic, mantissa)

<u>I</u> is a fixed point integer type item (i.e., binary point assumed to be after the least significant bit).

<u>H</u> is a binary coded Hollerith item (i.e., each character is a six bit code).

<u>A</u> defines a fixed point number with a fractional part (i.e., a binary point is within the item).

S is a status type item. (i.e., each value of the item stands for a unique status.)

#BITS is the total number of bits an item contains (including sign, if it is a signed item). (For the JOVIAL Interpreter, this must be less than or equal to 36 bits.)

S/U designates that the item is signed or unsigned (i.e., it can or cannot take on negative values.)

#CHARACTERS is the number of characters a BCH type item contains. The total number of bits in this type of item is 6 times the #CHARACTERS. (For the JOVIAL Interpreter the number of characters must be less than or equal to 6.)

#BITS RIGHT OF BINARY POINT specifies, for A type items, the accuracy desired for the particular item.

STi is the status, given in one to six alphanumeric characters.

$ signifies the end of the Item Declaration Statement. It should be noted that the order of the elements within an item definition is important and cannot be changed. Between any two elements in a statement, at least one blank must exist. (There may be more than one blank, if desired).

2.1.1 Parameter Items: It is sometimes convenient to assign items a value prior to the operation of a program. In effect, this permits the use of a constant throughout a program by use of a symbolic reference. This is done by adding the letter "P", followed by one or more blanks and the value of the item in the coding type specified by the Item Declaration. (See Section 3. "Types and Uses of Constants".)

Examples:

```
ITEM PI F P 3.1416$
ITEM ABC H 4 P 4H(HELP)$
ITEM XYZ I 7 U P 73$
ITEM PIFIX A 17 U 15 P 3.1416A15$
```

2.2 Table Definition

Note: The following pertains to "fixed length entry" tables only. "Variable-length entry" tables will be discussed only in Section 11.12.

Section 2.1 described the technique for defining individual items. In effect this permits the definition of single-valued quantities unlike any other quantity in the program. The majority of items in use, however, are usually not this unique type. More frequently, an item is part of a table, which is a collection of items. One entry of a table contains one repetition of all the items in the table. There may be any number of entries in a given table. Thus the number of entries determines the number of repetitions of each item in the table.

When one wishes to define a table, he must be able to define the above situation. He must be able to name all the items in the table and then give the number of repetitions of these items in this table. One further piece of information is necessary. This pertains to the dynamic status of the number of entries. Either it is fixed at all times, or it varies.

2.2.1 Table Declaration Statement

All of the preceding is specified in the Table Declaration Statement. The format for Table Declaration Statements is as follows:

```
TABLE  NAME  V/R  #ENTRIES$
BEGIN  ITEM...$  ITEM...$ ...END
```

TABLE declares this to be a Table Declaration Statement.

NAME is the field which contains the name of the table. This may be two to six alphanumeric symbols, the first of which must be a letter.

V means that the table has a variable number of entries. This type of table will always have a control word appended to it which contains the current number of entries and number of words in the table.

R means that the table has a fixed number of entries.

#ENTRIES is the number of entries for a fixed length table or the maximum number of entries for a variable length table.

Following the Table Declaration Statement is a list of all items to be contained in the table. These are preceded by a "BEGIN" and followed by an "END". All item definitions are as described in section 2.1. Every table definition must be followed by at least one item definition enclosed by the brackets "BEGIN" and "END". Tables without items have no meaning in JOVIAL.

2.2.2 Definition of Tables Identical to Previously Defined Tables

It sometimes happens that one knows that he will have two or more identical sets of data to which his program refers. Assuming that one of these sets (tables) is defined in the Compool (or by the program), it is somewhat awkward to have to redefine a completely new, but identical, table. The ability to describe this situation has therefore been put into the JOVIAL language. The following statement will permit the automatic definition of a table and all its items. (For this example, we assume TAB has been defined previously.)

```
TABLE TABΔ R 1 L;$
```

Δ may be any letter or number. L states that we are defining a table to be identical with the table whose name consists of the first n-1 characters of the n characters given in the NAME field of this statement (in this case TAB). All items will then automatically be defined for this new table by adding the character Δ to the name of every item and copying the properties of the respective items.

The number "1" in this example states that there is 1 entry in the table TAB Δ; "R" means that it is a fixed-length table. If these two fields are left blank, it will be assumed that the new table has the same number of entries and the same "V" or "R" as the originally defined one. If either field is to be included, both must be present. To refer to any item within the table TAB Δ, the original item name plus the character Δ must be used.

Example: Assume that table ABC has been defined in the following fashion:

```
TABLE ABC R 5Ø$
BEGIN ITEM XYZ F$
      ITEM QRS I 7 U $ END
```

The following table definition card can be used:

```
TABLE ABCD R 2 L$
```

It has the same effect as the following:

```
TABLE ABCD R 2$
BEGIN ITEM XYZD F$
      ITEM QRSD I 7 U$ END
```

One of the big advantages of this feature is that the definition of the new table will change automatically when the original one changes (in the Communication Pool or Program).

2.2.3 Definition of Tables of Constants

In section 2.1.1, it was shown how to assign values to items not belonging to tables. The present section demonstrates the technique for assigning constant values to items which are within tables.

The following illustrates the method with an example:

```
TABLE ABC R 5$
BEGIN  ITEM XYZ F$
       BEGIN 13.6 14.Ø 1.7 13.9 15.36$ END
       ITEM PQR H 2$
       BEGIN 2H(AB) 2H($$) 2H(.1) 2H(3A)
             2H (99)$ END
       ITEM MN H 6$ END
```

All items which are to contain constants have their definition statements followed by the list of constants. The list of constants is preceded by the bracket "BEGIN" and followed by a "$" and the bracket "END". Note that all items in the table do not have to be constants. If there are fewer constants in a list than there are repetitions of the item, the non-specified values will be set to Ø.

3. TYPES AND USES OF CONSTANTS

When programming in JOVIAL, constants are used in a similar fashion to items which are not in tables. Except for the fact that items can be set and constants cannot, constants and items are manipulated in much the same way.

The format for an item name is two to six arbitrary alphanumeric characters. This set of characters has no meaning in itself. It is simply a symbol which represents a particular number. The format for constants, on the other hand, depends on the type of constant being used. In short, the characteristics of a constant are described in the writing of the constant. Corresponding to the five types of items, there are five types of constants. These are described in the following.

(1) Floating point: A floating point constant consists of a sign (optional), a set of numbers, and a decimal point.

Examples: .ØØ156 -15.35 +17ØØ. 1.

(2) Fixed Integers: A fixed integer consists of a sign (optional), and a set of numbers.

Examples: 13 +1 -17ØØ

(3) Fixed Numbers with Fractional Parts: This type of number consists of a sign (optional), a set of numbers, a decimal point and the letter "A" followed by an integer specifying the required number of binary bits to the right of the binary point (the "accuracy" of the item).

Examples: .ØØ156A18 -14.25A2 +17ØØ.1A4

(4) Hollerith constants: These have the following format: nH(cc..), where n is the number of characters within the parentheses, and the c's are any legal Hollerith characters.

Examples: 2H(AB) 4H(ABCD) 6H($$A1,B)

The number of characters within the parentheses must be exactly equal to the number specified preceding the "H". (For the JOVIAL Interpreter, this number cannot exceed 6.)

(5) Status Value Constants: These constants are used only when setting or testing status items. Their format is described in the following examples:

V(GOOD) V(FAIR) V(ENEMY)

The number of characters within the parentheses can not be larger than 6.

3.1 Expressing a Power of 1∅

It is sometimes convenient, particularly with very large or very small numbers, to express the constant as a coefficient and a power of 1∅. The letter "E" followed by a positive or negative integer may be used to express the power of 1∅.

Examples: 13.1E3 13.1E-3 13.1A5E-3 13.1E3A5

The examples are equivalent to:

131∅∅ floating, .∅131 floating, .0131 fixed with 5 bits to the right of the binary point. An optional representation for the symbol "E" is the symbol "$$". Either may be used.

4. ARITHMETIC

This section will discuss some of the rules of arithmetic without attempting to describe how the arithmetic is used in the language. In JOVIAL, arithmetic is never done as an end in itself. It is used only as an instrument for setting some value or making a decision. Thus this discussion of arithmetic by itself is somewhat out of context. However, since some basic rules exist which apply no matter how the arithmetic is used, it is felt that a separate discussion has some value.

4.1 The Basic Operators +, -, *, /

As one would expect, the four basic arithmetic operations are addition, subtraction, multiplication, and division. These operators, in combination with a set of variables and constants, comprise what is called an arithmetic expression. (In this document single terms, such as a constant or a variable, will also be considered to be arithmetic expressions.)

4.1.1 Expressions of almost any complexity are allowed

```
Examples:  ABC*EFG+HIJ/KLM
           ABC-(EFG+HIJ)-KLM
           ABC*((BCD+EFG)/(MNO-PQR)+ABLE)
```

One type of expression is not legal at present. This is an expression completely enclosed in parentheses and preceded by a "+" or "-" sign.

4.1.2 No implied multiplication is allowed

Every multiplication must be represented by the symbol "*".

```
Example:  (AB + BC) * (EF + GH)
```

4.1.3 Precedence of arithmetic operations

When two or more operations are involved in one arithmetic expression, it is at times necessary to know which will take precedence (e.g., be operated first, second, etc.) in order to write the expression properly. As an example of ambiguous expressions consider the following:

```
   ABC/CDE/GHI
ABC/CDE + FGH
```

The rules may be summarized in the following fashion:

(1) All operations within parentheses are performed prior to any not within parentheses. Those most deeply imbedded in parentheses will be performed prior to

those less deeply imbedded.

Example: AB+((CD+EF)+GH)

In the above expression, the order of operation would be the following:

(1) CD + EF→ R1
(2) R1 + GH→ R2
(3) AB + R2→ R3

(2) Once parentheses have been eliminated (or none existed), multiplication and division precede addition and subtraction.

Examples: (a) AB + CD * EF

The order for carrying out the above example is the following:

(1) CD*EF→ R1
(2) AB+R1→ R2

(b) ABC/(CD+EF) + GH

The order for this example is:

(1) CD+EF→ R1
(2) ABC/R1→ R2
(3) R2+GH→ R3

(3) Once parentheses have been eliminated (or none existed) operators on the same level will be performed proceeding from left to right. (*, / are on the same level. +, - are on the same level).

Examples:

(a) AB/CD/EF
This is performed in the following fashion:
(1) AB/CD→ R1
(2) R1/EF→ R2
In other words, AB/CD/EF is equivalent to AB/(CD*EF).

(b) AB/CD*EF
This is carried out in the following fashion:
(1) AB/CD→ R1
(2) R1*EF→ R2
In otherwords AB/CD*EF is equivalent to AB/(CD/EF).

4.1.4 Mixing of different types of items and constants in an expression:

(1) Status Items and Status Constants may never be used in an arithmetic expression.

(2) Floating items or constants may never be used in the same arithmetic expression as fixed items or constants.

4.1.5 The accuracy of fixed point arithmetic calculations:

A frequent problem is the determination of the significance of the result of a fixed point arithmetic calculation. If one wishes to determine this in JOVIAL, he can best tell by looking at each operator and its two operands, since rules govern the significance of each individual arithmetic operation.

These rules are:

(1) Whenever an operation between two integers takes place, the result will be an integer.

Examples: 3*7 results in the integer 21
7/3 results in the integer 2
3/7 results in the integer Ø

(2) When a calculation involving an integer and an "A" type item or constant takes place, the result will have the same accuracy as the "A" type item.

Example: 3*7.1A2+6 produces the result 27.3A2
3.ØA6/7 produces the result .429A6

(3) When a calculation involving two "A" type values takes place, the result will have the same significance as the least significant "A" type item.

3.25A3*3.5A4-1.ØA5 produces the result 10.375A3.
6.25A3+1.ØAØ produces the result 7.ØAØ

4.2 Exponentiation

Another arithmetic operation built into the language is exponentiation (i.e., taking an arithmetic expression to a power which is an arithmetic expression).

When writing expressions containing exponents for mathematical texts, one has a decided advantage, since he has at his disposal two dimensions. Thus he can write as follows:

$$AB^2+BC^{CL^{(EF+GH)}}$$

Given a one dimensional IBM card, however, one must, in a sense, say when he is going up (and down). Thus the brackets "(*","*)" are used, and the above expression is written in the following fashion in JOVIAL:

```
AB(*2*)+BC(*CD(*(EF+GH)*)*)
```

Note that there must be an equal number of up and down brackets. Any legal arithmetic expression may be used in the exponent. A certain amount of mixing may be done in exponentiation. An integer value may be used as the exponent of a floating expression. Example: AB(*2*), where AB is a floating item.

4.3 Taking the absolute value

It is frequently necessary to take the absolute value of an item or arithmetic expression. Thus one might want to say:

$$|ABLE| \quad \text{or} \quad |ABLE^2 - BAKER^2| - |CHARLY\text{-}DOG|$$

These expressions are written in JOVIAL in the following fashion:

```
ABS(ABLE)
ABS(ABLE(*2*)-BAKER(*2*))-ABS(CHARLY-DOG)
```

5. SETTING ITEMS (The Assignment Statement)

Note: A good deal of what follows pertains to subscripts as well as items although the following will discuss items only.

A considerable portion of programming consists of the setting of items. In JOVIAL, there is one basic format for the setting of items. This format is called the Assignment Statement. It has the following structure:

```
LEFT TERM (ITEM TO BE SET)    ASSIGNMENT OPERATOR "="    RIGHT TERM $
```

The RIGHT TERM may be an arithmetic expression, and the symbol "$" represents the end of the statement.

The following are legal Assignment Statements. They will accomplish the setting of the item ABLE to the value of the right term.

```
ABLE =  BAKER $
ABLE =   -3 $
ABLE =  3* (BAKER - ABLE) $
ABLE =  ABLE (*2*) + 1$
```

5.1 Use of the Assignment Statement for Fixing and Floating

It will be recalled that in Section 4.1.4 it was stated that one could not mix fixed and floating numbers in an arithmetic expression. Therefore all numbers in the right term of the assignment statement must be either of a fixed or floating variety.

On the other hand, the left term (the item being set), does not have to agree with the right term. In other words, if the right term is a floating item or produces a floating result, the left term may be defined as fixed (either "I" or "A" types) and this will produce instructions to fix the result prior to storing it in the left term. The converse is true for a fixed to floating conversion. Thus the assignment statement may be used for fixing and floating when these functions are required.

5.2 Significance of Fixed Point Results

As has been pointed out in section 5.1, the left term of the assignment statement governs, in the final analysis, what should be done with the right term. This applies to the final scaling of fixed point right terms prior to storing them in fixed point left terms. Therefore, once the result of the right term has been calculated, an examination is made of the left term. If the left term has more places to the right of the point than the result, trailing zeros are inserted following the least significant bit of the result prior to storing into the left term.

If the left term has fewer places to the right of the point than the right term, the right term is rounded prior to storing into the left term.

Examples:

Define the following item:

```
ITEM ABLE A 1Ø U 3 $
```

(i.e., ABLE has 3 bits to the right of the binary point).

The following assignment statements produce the stated results.

(a) ABLE = 5.5A1$ Note: 5.5A1 = $(101.1)_2$
ABLE gets set to $(\text{ØØØØ1Ø1.1ØØ})_2$.

(b) ABLE = 1Ø.875 A4$
ABLE gets set to $(\text{ØØØ1Ø1Ø.111})_2$.

(c) ABLE = 1Ø.4375A4$
ABLE gets set to $(\text{ØØØ1Ø1Ø.1ØØ})_2$.

6. A DISCUSSION OF STATEMENTS AND STATEMENT LABELS

6.1 Properties of Statements

In the preceding sections, a good deal of reference has been made to statements. So far Item Declaration, Table Declaration, and Assignment Statements have been discussed. Statements represent the main means of program writing in JOVIAL. The format of the particular statement is a function of its purpose. In the case of Assignment Statements and some others to be discussed later, one can describe in general the format of the statement. In other cases a general description of format is quite difficult.

However, certain properties exist for all simple statements. (Note: The present discussion refers only to what are called simple statements. Compound statements will be discussed in Chapter 9.)

(1) Every statement has an operator or a declarator of some kind. In the case of the Item and Table Declaration Statements, the words "ITEM" and "TABLE" were the declarators. In the assignment statement the symbol "=" was the operator.

(2) Every statement must end with the symbol "$". Other than the above two comments, nothing can be said about the general nature of all statements. As each operator and declarator is discussed, the format for its particular statement will be discussed.

6.2 Labeling Statements

A facility exists for naming any statement used in a program. The need for labeling arises only when a reference is made to the statement by another statement. In the current version of JOVIAL, a reference would be made to another statement only for the purpose of changing the sequence of control in the program. For example, no reference to a statement labelwould ever be made within an Assignment Statement. However, the Assignment Statement, and most of the other statements to be discussed, may themselves be labelled (e.g., given a name).

The format of a statement label is the same as that for item and table names. That is, it consists of two to six alphanumeric symbols, the first of which must be a letter. Indeed, one of the differences between JOVIAL and machine language programming is that a variable name and a statement label used by the same program may be the same symbol. This does not imply that one should strive to use the same names for variables and statement labels, but it does point out that the treatment of variables in JOVIAL is considerably different than their treatment in machine language programming.

Once a symbol has been decided upon for a statement label, the statement is labeled by preceding it with the label followed by the symbol "." (period).

Examples:

```
A36.  ABLE = 17$
ZZY2. ABLE = 3*BAKER $
AMMONA.BAKER = 4*(ABLE +36)$
```

A particular symbol may not appear as a statement label more than once in the main program. (The main program does not include Procedures, to be discussed in Chapter 16).

7. GOTO STATEMENTS - Unconditional Change of Control

Given a sequence of Assignment Statements, control would proceed from one to the other in the order they are presented. Frequently it becomes necessary to change this "normal" sequence unconditionally. One means of accomplishing this is a GOTO statement. The format of the GOTO statement is shown in the following examples:

```
GOTO  A36$
GOTO  ZZY2$
GOTO  AMMONA $
```

Note that the symbol "." is not used when the statement label is used as the object of a GOTO statement.

Only statement labels can be used as objects of GOTO statements.

These labels must be used somewhere in the program to name a statement. In brief, a GOTO statement must have someplace to go.

GOTO statements can themselves be labeled, as can any statements which may be referenced by other statements in the program.

```
Examples:  A1.      GOTO  ABIE3$
           AYX.     GOTO  A1$
           DESK.    GOTO  DEN$
```

8. DECISION MAKING - IF Statements

8.1 Conditions

All decisions in programming are based on the truth or falsity of some condition or combination of conditions. A condition (Ci) has three parts which are arranged in the following format:

LEFT TERM   RELATIONAL OPERATOR   RIGHT TERM

where; (1) the LEFT TERM is an arithmetic expression
(2) the RELATIONAL OPERATOR is one of the following:

(a)	EQ	equal
(b)	NQ	not equal
(c)	GR	greater
(d)	GQ	greater than or equal
(e)	LS	less than
(f)	LQ	less than or equal

(3) the RIGHT TERM is an arithmetic expression which is the same type as the left term (e.g., fixed or floating).

Examples:

```
ABLE  EQ  BAKER
A13B  NQ  BABY + 1
3*ABLE  GR  BAKER-7
ABLE (*2*) + BAKER(*2*)  GQ  CHARLY(*2*)
13.6 * QRS + 3.  LS  PGI + 17.6
SQUARE /  CIRCLE  LQ  TRIANG(*2*)
```

8.2 IF Statements

The main decision making statement in JOVIAL is the IF Statement. The IF statement consists of the operator "IF", followed by at least one condition or combination of conditions (see 8.2.2) and is followed by the symbol "$" to denote the end of the statement.

Function of the IF Statement

In the event the situation being tested is true, control proceeds to the next statement. In the event it is false the next statement is skipped and control proceeds from there.

8.2.1 The Simplest IF Statement

The simplest (and most frequently used) IF statement contains one condition. Thus it has the following basic format:

IF C$ where C is a condition as described in Section 8.1.

Examples:
```
IF  ABLE  EQ  BAKER$
IF  A13B  NQ  BABY +1$
IF  3*ABLE  GR  BAKER-7$
```

8.2.1.1 Some Programming Examples

The following are some problems and possible JOVIAL programs to solve them.

Note: The STOP Statement (see Chapter 10) will be used to denote completion of the problems.

(1) Given item NUMB defined as a fixed point integer. compute the factorial of NUMB and store it in the integer item FACT.

```
          FACT = 1$

    A1.  IF NUMB LQ 1$
         STOP$
         FACT = FACT*NUMB$

         NUMB = NUMB-1$

         GOTO A1$
```

(2) Given the floating item SQUAR, use Newton's Method to compute the square root of SQUAR and store it in the floating item SQRT. Continue the iterations until $|SQRT - SQUAR| < .ØØØ1$.

```
         SQRT = .5*SQUAR$
      A2.IF ABS(SQRT(*2*)-SQUAR) LS .ØØØ1$
         STOP$
         SQRT = .5*(SQRT+SQUAR/SQRT)$
         GOTO A2$
```

8.2.2 More Interesting IF Statements - Logical Operators

Section 8.2.1 discussed decisions made on the basis of one condition. Frequently a decision must be made on the basis of more than one condition. Also, it is sometimes convenient to express a decision making statement as the negation of a condition or set of conditions. These latter two functions may be accomplished with the IF Statement and the logical operators AND, OR, and NOT. Some illustrative examples will now be given to demonstrate the possible uses of these operators.

(1) A Decision based on the simultaneous Truth of Two or more conditions - Logical Operator AND

If one wishes to perform some function based on the simultaneous truth of two conditions, the format for the IF Statement is:

IF C_1 AND C_2\$ where C_1 and C_2 are the conditions.

If either C_1 or C_2 (or both) are false, the statement will result in a false transfer of control (i.e., the next statement will be skipped). If both C_1 and C_2 are true, the IF Statement is true, and the next statement will be executed.

Examples:

The sequence of statements:

```
        IF ABLE EQ 3*BAKER AND TPOS NQ 17.6$
        BAKER = 15$
        ABLE  = 17$
```

is equivalent to the following sequence:

```
        IF ABLE EQ 3* BAKER$
        GOTO A1$
        GOTO A2$

    A1. IF TPOS NQ 17.6$
        BAKER = 15$

    A2. ABLE  = 17$
```

The sequence of statements:

```
        IF TIDY EQ V(HOSTIL) AND POSN EQ V(NEWYRK)$
        GOTO B1$
        ALARM = Ø$
        STOP$

    B1. ALARM = 1$
        STOP$
```

is equivalent to the following sequence:

```
        IF TIDY EQ V(HOSTIL)$
        GOTO C1$
        GOTO C2$

    C1. IF POSN EQ V(NEWYRK)$
        GOTO C3$

    C2. ALARM = Ø$
        STOP$

    C3. ALARM = 1$
        STOP$
```

The use of the operator AND can be extended to testing the simultaneous truth of three or more conditions. A general example is:

```
        IF C1 AND C2 AND C3...AND Ci...$
```

(2) Decisions Based on the Truth of at Least one Condition - Logical Operator OR

The performance of a particular statement or set of statements might be predicated on the truth of any one (or more) of a set of conditions. The form of an IF statement which makes this decision is:

```
IF $C_1$ OR $C_2$ OR $C_3$...$
```

In this statement the truth of the conditions are tested (from left to right) until one is found that is true. When this occurs, control proceeds to the next statement. If no true condition is found, control skips one statement and proceeds from there.

Examples:

```
                                          LS
(a)   IF TIDY EQ V(HOSTIL) OR ALT IS 1ØØ$
      GOTO C1$
      ALARM = Ø$
      STOP$
C1.   ALARM = 1$
      STOP$

(b)   IF XY/2.1 LS ABLE (*2.+BAKER*) OR MN EQ 3
      OR CHARLY GR 17.6*DOG$
      GOTO D1$
```

(3) The Use of AND and OR in one statement

A decision may be based on any combination of conditions and the operators AND and OR. There will be no attempt to give all the rules for these combinations. However, a study of elementary Boolean Algebra should suffice to prepare the reader for any situation.

Example:

```
IF $C_1$ AND ($C_2$ OR $C_3$ )$
GOTO TRUE$
GOTO FALSE$
```

is equivalent to:

```
IF $C_1$ AND $C_2$ OR $C_1$ AND $C_3$$
GOTO TRUE$
GOTO FALSE$
```

is equivalent to:

```
    IF $C_1$$
    GOTO E1$
    GOTO FALSE
E1. IF $C_2$$
    GOTO TRUE$
    IF $C_3$$
    GOTO TRUE$
```

(4) <u>Testing the Negative of a Condition or a Set of Conditions- The Operator NOT</u>

It is sometimes desirable to express an IF statement in terms of the negative of a condition or a set of conditions. The operator NOT is used for this purpose. Again, a complete set of rules for this operator will not be given here. A few examples will suffice:

(a) IF NOT ABLE EQ 3$

is equivalent to:

IF ABLE NQ 3$

(b) IF NOT (ABLE EQ 3 AND BAKER GR 4)$

is equivalent to:

IF ABLE NQ 3 OR BAKER LQ 4$

(c) IF NOT (ABLE LQ 3 OR BAKER GQ 4)$

is equivalent to:

IF ABLE GR 3 AND BAKER LS 4$

(d) IF ABLE EQ 3 AND NOT (BAKER EQ 4 OR NOT BAT LS 6)$

is equivalent to:

IF ABLE EQ 3 AND (BAKER NQ 4 AND BAT LS 6)$

9. COMPOUND STATEMENTS

9.1 Definition of Compound Statements

A compound statement is a group of consecutive simple statements preceded by the left bracket "BEGIN" and followed by the right bracket "END".

Example:

```
BEGIN  ABLE = 3$
      BAKER = 4$
   IF XY EQ = 7$
         MP = 4$
         NO = Ø$  END
```

9.2 A Compound statement may be labelled

The label must follow the bracket "BEGIN" and precede the first simple statement of the compound statement.

Example:

```
BEGIN A1.  INT = 3$
           IND = 1$
           RST = 3$  END
```

9.3 Compound statements may be nested

Any number of compound statements may appear on the same (or different) levels within another compound statement. Care must be exercised to have a bracket "END" for every bracket "BEGIN".

Example: (Assume Si's are simple statements)

```
 BEGIN       S1$
             S2$
              .
              .
             Sn$
      BEGIN  Sn+1$
              .
              .
             Sn+m$
      END
             Sp$
      BEGIN  Sp+1$
              .
              .
             Sr$
        BEGIN
             Sq$
              .
              .
             St$
        END
     END
END
```

9.4 Any simple or compound statement within a compound statement may be labelled.

A labelled statement within a compound statement may be transferred to from inside or outside the compound statement.

Example:

```
        IF ABLE EQ 3$
        GOTO B1$
     BEGIN   S1$
        B2.  S2$
        B1.  S3$
             S4$
             GOTO B2$

     END
```

(Note: The above example merely illustrates the ability to go to simple statements within compound statements. The particular sequence of statements is not necessarily a meaningful one.)

9.5 Use of the compound statement following the IF Statement

The compound statement permits one to execute more than one simple statement in the event an IF Statement is true. This follows from the fact that the truth of an IF Statement will cause the execution of the next statement (simple or compound), and the IF Statement being false will cause control to skip the next complete statement (simple or compound). Thus in the event an IF Statement is false, and the statement immediately following is compound, the complete compound statement is skipped.

Example:

```
(a)      IF ABLE EQ 7$
    BEGIN RST = 6$
          XYZ = 3$
          STOP$
    END   RST = 7$
          XYZ = 4$
          STOP$
```

In this example, if ABLE equals 7 RST will be set to 6 and XYZ to 3. If ABLE doesn't equal 7 RST will be set to 7 and XYZ to 4.

```
(b)  IF ABLE - BAKER LS 1.$

BEGIN ALARM = ∅$
      DOG = 3$
      IF ABLE-BAKER LS .5$

      BEGIN
      CAT = 7$
      KITTEN = 12$
      END
       STOP$
END    ALARM = 1$
       STOP$
```

In this example, if the difference between ABLE and BAKER is less than 1, the item ALARM is set to ∅, and the item DOG is set to 3. If the difference is also less than .5, then CAT is set to 7 and KITTEN to 12, prior to stopping. If the difference were greater than or equal to 1, then ALARM would be set to 1 prior to halting, and none of the other items would be set.

9.6 Final Remarks on the Compound Statement

There are, in addition to compound statements, other places where the brackets "BEGIN" and "END" must be used. Because of this, experience has shown that a liberal use of these brackets leads to some mild confusion regarding which BEGINS's and END's are mates. If this becomes the case, or anyone has a particular aversion to these brackets, their use with compound statements may be avoided. One example will suffice to demonstrate the means of avoiding the use of compound statements following IF statements.

Example:

```
        (a)  IF ABLE EQ BAKER $
             BEGIN  S1$
                    S2$
                    S3$
                    S4$  END
             STOP$
        (a)  May be rewritten as (a1)
        (a1) IF ABLE NQ BAKER $
             GOTO A1$
             S1$
             S2$
             S3$
             S4$
        A1.  STOP$
```

10. STOP STATEMENTS

The operator STOP is used when it is desired to make the object program come to a halt. In other words, the STOP statement is replaced by (or is interpreted as) the machine language HALT instruction.

The simplest format of the STOP statement is the operator STOP followed by the symbol "$".

Example: STOP $

Another format permits a statement label to be inserted between the operator and the symbol "$". The effect will be to permit a transfer to the statement with this label if the "CONTINUE" button is depressed after the halt takes place. (At present, this feature does not operate in the JOVIAL Interpreter System.)

Examples:

```
STOP A1$
STOP BC74 $
```

11. SUBSCRIPTS AND INDEXING

The preceding sections have used only non-subscripted items in all examples. These are items which do not belong to tables, or items which belong to one entry tables. As mentioned previously in the section on Item and Table Definition, the majority of references are to items which are members of tables. To refer to one of the latter items, the position (index value) of the item within the table must be appended to the item name. This position is either a number or a single letter, or has the format letter plus or minus a number It is called a subscript.

Subscripts when used as index values for items are positive integers having zero as the minimum quantity which they may attain. (This does not mean that a letter subscript may not take on negative values. The letter I when used as a part of the subscript I + 6 may take on the values -1, -2, -3, -4, -5, and -6.) The subscript stands for the relative position in the table to which an item belongs.

When used as an index value, the subscript immediately follows the item being indexed and is enclosed in the brackets "($" AND "($" and "$)".

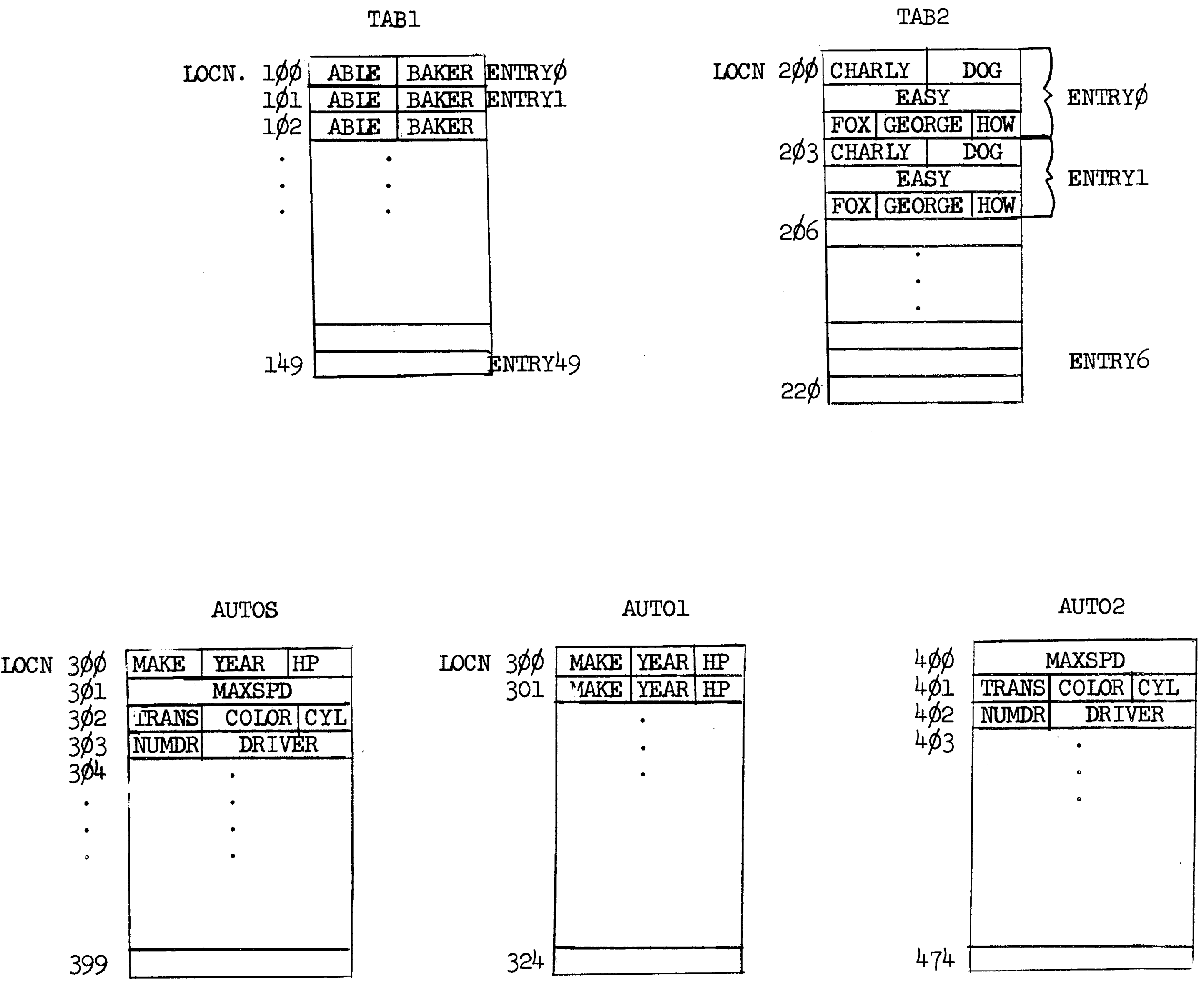
Examples:

```
ABLE($2$)
BAKER($J$)
CHARLY($I-4$)
DOG($K+3$)
```

11.1 Some Examples of Tables

For the ensuing discussion, it will be helpful to describe some hypothetical tables which can be used to illustrate the various points which are made. Figure 1 will be used for this purpose. Throughout the remainder of this section, it will be referenced for examples.

FIGURE 1


TAB1
LOCN. 1ØØ
1Ø1
1Ø2
149
ABLE
BAKER
ENTRYØ
ENTRY1
ENTRY49
TAB2
LOCN 2ØØ
2Ø3
2Ø6
22Ø
CHARLY
DOG
EASY
FOX
GEORGE
HOW
ENTRYØ
ENTRY1
ENTRY6
AUTOS
LOCN 3ØØ
3Ø1
3Ø2
3Ø3
3Ø4
399
MAKE
YEAR
HP
MAXSPD
TRANS
COLOR
CYL
NUMDR
DRIVER
AUTO1
LOCN 3ØØ
301
324
MAKE
YEAR
HP
AUTO2
4ØØ
4Ø1
4Ø2
4Ø3
474
MAXSPD
TRANS
COLOR
CYL
NUMDR
DRIVER

11.2 Constant Subscripts

Sometimes specific members of a table must be referenced. This implies that the relative position of the item being used is known and does not vary. In this case the subscript within the brackets is a constant.

Examples:

Using Figure 1, the following is a table of item references in JOVIAL and the effective addresses which would be generated by these references.

Reference	Address
ABLE ($Ø$)	1ØØ
BAKER(5)	1Ø5
ABLE (49)	149
DOG ($Ø$)	2ØØ
FOX ($Ø$)	2Ø2
EASY (3)	21Ø
GEORGE(6)	22Ø

Note that the number of words per entry has no effect on the subscript.

The subscript always represents the entry number. Therefore if one is interested in an item in the kth relative position of a table, he need only specify k and need not be concerned with the number of words per entry in the table.

11.3 Variable Subscripts

The most frequent use of subscripts is for cycling through many entries in a table, that is, repeating the same operation a number of times, changing the entry number of the items being used in the operation on every repetition. For this and other reasons it is necessary to use symbolic subscripts. Symbolic subscripts have two forms. One is a single letter, the other a single letter plus or minus a constant.

```
Examples:   ABLE ($A$)
            BAKER ($D$)
            CHARLY($I$)
            DOG ($Z$)
            EASY ($J+1$)
            FOX  ($Y-3$)

            GEORGE($K-4$)
```

Again, the meaning of the subscript is entry number. The means by which subscripts are set and modified will be discussed in Section 11.4.

The following table presents some subscripted variables, the subscript values, and the effective address derived from the latter, based on Figure 1.

ITEM	SUBSCRIPT VALUE	EFFECTIVE ADDRESS
ABLE (I)	I = ∅	1∅∅
BAKER (J)	J = 1	1∅1
BAKER (K)	K = 49	149
BAKER ($L-1$)	L = 1	1∅∅
ABLE ($M+6$)	M = 4∅	146
CHARLY (N)	N = 1	2∅3
FOX (O)	O = 5	217
EASY (P-3$)	P = 4	2∅4
GEORGE ($Q + 2$)	Q = 4	22∅

11.4 Setting, Modifying and Testing Subscripts - The FOR Statement

In many respects, symbolic subscripts are like integer type items. In fact, it will be seen later that as long as certain rules are followed, subscripts can be manipulated exactly as items are.

However, certain major differences between items and subscripts exist. First of all, subscripts are not defined in the Communication Pool or with Item Declaration Statements. Secondly, the definition of a subscript is done with a particular statement, called the FOR Statement. Lastly, the definition of a subscript holds only for the one statement (simple or compound) immediately following the FOR Statement (unless the next statement is a FOR statement). Once this statement is complete, it is illegal to use the subscript in the following statements unless it is defined by another FOR statement.

11.4.1 The Incomplete FOR Statement

There are two forms of the FOR Statement, each having a different purpose. The incomplete form is used merely to assign a subscript a single value. Its format looks like the Assignment Statement's (Section 5) with the operator FOR preceding it and the left term always a one letter subscript.

Examples:

```
FOR I = 6$
FOR J = ∅$
FOR A = BAKER$
FOR B = CHNNUM/2+1$
```

Some programming examples:

```
(a)  FOR I = 1$
     MAKE($I$) = V(FORD)$
```

(Note: The above is not an example of good programming. It would have been more efficient to make the following statement: MAKE (1) = V(FORD)$

```
(b)  FOR J = ABLE$
     IF HP($J$) LS 15ϕ$
     GOTO B1$

(c)  FOR K = BAKER*3/2$
     BEGIN IF HP ($K$) LS 15ϕ$
           MAXSPD ($K$) = 6ϕ.ϕ$ END
```

11.4.2 The Complete FOR Statement

The purpose of the Incomplete FOR Statement was simply to set a subscript to be used by the following statement. The complete FOR Statement is used to generate a loop, that is, cause the next statement to make a given number of iterations, each time modifying the subscript by a given amount.

The general format of this statement is the following:

```
FOR S = A,B,C$
```

where; S is the subscript.
A is the initial value (arithmetic expression).
B is the amount by which to modify the subscript for each iteration (plus or minus a constant or variable).
C is the final value (arithmetic expression).

Examples:

(a) Clear all items ABLE in table TAB1 (See Figure 1).

```
FOR I = ϕ,1,49$
ABLE ($I$) = ϕ$
```

Alternately,
```
FOR I = 49, -1, ϕ $
ABLE ($I$) = ϕ$
```

(b) Set all items FOX in table TAB2 to - ϕ

```
FOR J = ϕ,1,6$
FOX ($J$) = - ϕ$
```

Note that although there are three words per entry in table TAB2, the B-term is 1 and the C-term is 6. This illustrates again that subscripts are expressed solely in terms of entry number.

(c) Exchange the first ten pairs of items BAKER in the table TAB1.

```
        FOR I = ϕ,2,18$
BEGIN TEMP = BAKER ($I$)
        BAKER ($I$) = BAKER ($I+1$)$
        BAKER ($I+1$) = TEMP$   END
```

(d) Find an item FOX in the same entry as a DOG which equals 3 in table TAB2. Then starting with entry number FOX in table TAB1, set the BAKER's in TAB1 to Ø. If no such DOG exists, stop.

```
          FOR I = Ø,1,7$
      BEGIN IF DOG ($I$) EQ 3$
          BEGIN FOR J = FOX ($I$),1,49$
            BAKER ($J$) = Ø$
            STOP$
          END
      END        STOP$
```

11.4.2.2 Method of Modification and Testing

It is sometimes helpful to know the sequence of instructions which take place dynamically for a complete FOR Statement.

The general sequence is as follows:

(1) In place of the complete FOR Statement, an incomplete FOR Statement is produced.

```
FOR I = A$
```

(2) At the end of the definition of the subscript an assignment statement which modifies the subscript is inserted.

```
i.e., I = I +B$
```

(3) Immediately after the modification, the test instruction is inserted.

```
i.e., IF I EQ C+B$
      GOTO (STATEMENT FOLLOWING END OF LOOP)$
      GOTO (BEGINNING OF LOOP)$
```

Of course, if A and/or C had been arithmetic expressions, the necessary expansions would have been inserted prior to the assignment and/or test statements.

Example:
```
              FOR I=Ø,1,24$
            BEGIN A1.IF ABLE ($I$) EQ 3$
              BAKER ($I$) = 4$
            END
              A2. GOTO Z1$
```

The above example will have the same effect as the following set of instructions when it is interpreted (translated).

```
             FOR I = Ø$
      BEGIN A1.  IF ABLE($I$) EQ 3$
                 BAKER($I$) = 4$
                 I = I+1$
                 IF I EQ 25$
                 GOTO A2$
                 GOTO A1$ END
            A2.  GOTO Z1$
```

11.5 Using Subscripts as Variables

Once a subscript has been defined with a FOR Statement, it may be used both as an index value for items and as an integer type value. (i.e., It may be used as a variable in an arithmetic expression, or be used as the left term of an Assignment Statement, etc.)

An example of a sort program will serve to illustrate the use of subscripts in the fashion so described.

```
             FOR I = Ø,1,48$
      BEGIN  TEMP1 = I$
             A1.IF ABLE($I$) LQ ABLE($I+1$)$
             GOTO A2$
             TEMP2 = ABLE($I$)$
             ABLE($I$) = ABLE($I+1$)$
             ABLE($I+1$) = TEMP2$
             IF I EQ Ø$
             GOTO A2$
             I = I-1$
             GOTO A1$
      A2.    I=TEMP1$
      END
             STOP$
```

A subscript may be used in the right term of a FOR Statement. For example, the sequence:

```
      FOR I = Ø,1,1Ø$
      BEGIN FOR J = Ø,1,I-3$
                .
                .
                .
```

is a legitimate sequence of statements.

One further remark seems appropriate at this point. When a subscript is defined with a complete FOR Statement (i.e., The set, modify, and test instructions are to be automatically generated.), the value which will be modified and tested at the end of the loop is the value which remains after an Assignment Statement has been executed within the loop.

For example, the two statements following the FOR Statement will be executed only once in the following situation:

```
      FOR I = Ø,1,2$
      BEGIN ABLE($I$) = 3$
            I = I+2$
```

11.6 FOR Statements Defining Subscripts on the Same Level

In section 11.4, there was a sentence which stated that the definition of a subscript holds for one (simple or compound) statement which immediately follows the FOR Statement. The alert reader probably noticed the parenthesized expression immediately following this remark. It said "unless the next statement is a FOR Statement". The present section will attempt to explain this previously off-hand remark.

All examples given so far in which more than one FOR Statement was used (e.g., (d) in Section 11.4.2) have been examples of "loops within loops". That is, the second FOR Statement is contained in the compound statement following the first FOR Statement. A little reflection will show that the statement following the second FOR Statement will be repeated a total of (C2-A2+1) * (C1-A1+1) times, where the program appears as follows:

```
FOR I = A1,1,C1$
BEGIN FOR J = A2,1,C2$
     BEGIN
        S1
        S2
         .
         .
       END
END
```

In other words, the innermost loop (on J) is repeated $C_2 - A_2 + 1$ times for every one repetition of the outside loop (on I).

Frequently (particularly with single dimensional subscripting) it is desirable to have two (or more) subscripts used at one time within the "same" loop. One specifies the number of times a loop is to be repeated with a complete FOR Statement. Following this statement one (or more) FOR Statements (complete or incomplete) defining other subscript(s) may exist. In the statement immediately following the last FOR Statement in the sequence all defined subscripts may be used. At the end of the loop all subscripts which are defined with a complete FOR Statement will be modified. However, only one test will be generated. This test will be on the first subscript defined with a complete FOR Statement in the sequence. Thus, although more than one subscript may be set, used, and modified, only one test and thus "one loop" is generated.

Examples:

```
(a) FOR I = Ø,1,6$
    FOR J = Ø,2,12$
        S1$
    A2.  STOP$
    (a) is equivalent to the sequence (a').
    (a') FOR I = Ø$
         FOR J = Ø$
         BEGIN
              S1$
              J = J+2$
              I = I+1$
              IF I EQ 7$
              GOTO A2$
              GOTO S1$ END
         A2.  STOP$

(b) FOR J = Ø,1,17$
    FOR K = ABLE$
    FOR L = 53,-3,2$
  BEGIN S1$
        S2$
        S3$
   END
    A1.STOP

   (b) is equivalent to the sequence (b').
   (b') FOR J = Ø$
        FOR K = ABLE$
        FOR L = 53$
   BEGIN S1$
         S2$
         S3$
     L = L-3$
     J = J+1$
     IF J EQ 18$
     GOTO A1$
     GOTO S1$
   END
     A1.STOP$
```

It should be noted that (a') and (b'), which are legal examples, demonstrate that all the FOR Statements may be incomplete.

One further remark is that the order of incomplete and complete FOR Statements in sequence is immaterial. The only significance the order has is the designation of which subscript on which to test. This will always be the subscript which appears in the first (or the only) complete FOR Statement.

The following (C, C', and C") will produce equivalent results. In all cases the test will be made on K.

```
(C)  FOR I = HP$
     FOR J = NUMDR$
     FOR K = Ø,1,12$
     FOR L = 1,5,NONSNS$
BEGIN     .
          .
          .

(C') FOR J = NUMDR$
     FOR K = Ø,1,12$
     FOR I = HP$
     FOR L = 1,5,NONSNS$
BEGIN     .
          .
          .

(C") FOR K = Ø,1,12$
     FOR L = 1,5,NONSNS$
     FOR I = HP$
     FOR J = NUMDR$
BEGIN     .
          .
          .
```

Note that the C-factor of any complete FOR Statement after the first is never used. It must appear and be a legal expression. But it has no meaning.

11.7 Cycling Through Two or More Tables at one Time

The table AUTOS in Figure 1 contains the characteristics of a certain set of automobiles. It is assumed that the number of automobiles is fixed (25). Also, it takes 4 computer words containing 9 items to describe each automobile.

A program which might wish to set MAXSPD (maximum speed) to 60 mph if the horsepower (HP) is less than 90 and the year is earlier than 1947 or the number of cylinders is less than 6 might appear as follows:

```
FOR A = Ø,1,24$
BEGIN
IF HP($A$) LS 9Ø AND YEAR ($A$) LS 1947 OR CYL ($A$) LS 6$
MAXSPD = 6Ø.Ø$
END
```

Let's now assume that for some reason it is found necessary to break the table AUTOS into the two tables AUTO1 and AUTO2 of FIGURE 1.

Now the four words of information pertaining to each auto has been split into two parts, one containing three words and one containing one. The one further assumption will be that relative position has significance. Thus the information in entry number k of AUTO1 pertains to the same auto as the information in position k of AUTO2.

Using these assumptions, precisely the same program as was used to cycle through AUTOS may be used to perform the same function.

This is in general a property of the language. As long as relative position has significance, the actual distribution of the items in tables has no effect on programs which are written, and the same subscript may be used to cycle through two or more tables even if they don't have the same number of words per entry. Of course if items in the same relative positions in the different tables did not pertain to the same object, the use of one subscript for both would ordinarily not make much sense.

11.8 Variable Length Tables

The tables of Figure 1 were considered to be fixed length (e.g., R-type tables). This implies that the number of entries in the table has been decided upon prior to the program's assembly and to cycle through the complete table the same number of iterations is used every time the particular loop is operated.

The nature of variable length tables (V-type) on the other hand, is such that during the dynamic operation of the program the number of entries is subject to change. Thus at any given time it is desirable to have the number of entries (and number of words) available to the object program. All variable length tables are assumed to have these latter items of information preceding the first data word. Figure 2 illustrates a variable length table.

FIGURE 2

DISABL

499	NENT		NWDS
5ØØ	CHNNUM	REASON	DAY
5Ø1	EXTENT	REPLAC	
5Ø2	CHNNUM	REASON	DAY
5Ø3	EXTENT	REPLAC	
		°	
		°	
		°	

The table "DISABL" might be a file of all cars from table "AUTOS" (FIGURE 1) which have become disabled. This list might contain as few as no entries or, on particularly bad days, as many as 25.

11.9 The Modifiers NENT, NWDS, NWDSEN, ALL

Both fixed and variable length tables should be referenced in the same fashion in a JOVIAL program. Of course, when an entry is added or subtracted in a variable length table, the control word containing the number of entries must be changed accordingly by the JOVIAL object program. Since this never can occur with a fixed length table, one might say that their treatment is different in this respect. However, all other programming references should be identical.

Referring to variable and fixed length tables in a flexible fashion, ("flexible" meaning keeping the object program the same even though the table lengths change) requires some modifiers heretofore not discussed. The one most frequently needed is NENT, which stands for "Number of Entries".

11.9.1 NENT

We may well imagine the possibility that the number of cars in the table AUTOS might be increased or decreased, although this was presumably a fixed length table. If it were, any examples so far given in this text, which assumed the number of entries to be 25, would have to be modified, which as most programmers know, is a distasteful and dangerous practice. Thus a modifier is available which obviates the necessity of knowing or writing the exact number of entries. This modifier is NENT.

Like other modifiers of the language, the format for expressing NENT is similar to the standard mathematical notation for functions of something (e.g., f(x), F(Y), etc.). Thus immediately following the modifier NENT, the name of the table or an item of the table enclosed in parentheses is placed.

Examples:
```
NENT(TABLE)
NENT(ITEM)
NENT(AUTOS)
NENT(CYL)
NENT(HP)
```

When an item name is used following NENT, which is the preferred practice, it means the number of entries in the table containing this item.

For example, NENT(AUTOS) is equivalent to NENT(CYL) and NENT(HP).

The use of the modifier NENT should be used to program the example in Section 11.7, as follows:

```
FOR A = Ø, 1, NENT(YEAR) - 1$
BEGIN
IF HP($A$) LS 9Ø AND ...
```

Now if the number of entries in AUTOS is changed, the JOVIAL program is unaffected.

Whereas the use of NENT is highly desirable when referring to fixed length tables, in variable tables it is a necessity.

Thus, to cycle through all of the table DISABL, it is essential that one uses NENT as the limit of the loop.

```
Example:  FOR I = ∅,1.-NENT(DAY) - 1$
     BEGIN IF REASON($I$)  EQ V(ACCID)$
          REPLAC($I$) = 14$
     END
```

11.9.1.1 Using and modifying the number of Entries

The modifier NENT with its object item or table can be used and set as an integer.

```
Examples:
     ABLE = NENT(DOG) + 1$
     IF NENT (DISABL) GR 23$
     NENT(DAY) = NENT(DAY)+1$
```

When NENT is the left term of an Assignment Statement, both the number of entries(NENT) and the number of words (NWDS to be discussed in section 11.9.3) are set, the latter not requiring a reference in the JOVIAL object program. NWDS is set to the new number of entries times the number of words per entry.

11.9.2 The Modifier ALL

With the modifier NENT, one can write a FOR Statement to cycle through a complete table in the following fashion:

```
          A1. FOR I = ∅,1,NENT(TABLE)-1$
```

Since this is done frequently and is not convenient to write, the special modifier ALL has been added to the language for this purpose.

Thus the statement,

```
          B1. FOR I = ALL(TABLE)$
```

is equivalent to the statement A1 given above.

```
Examples:            FOR A = ALL(AUTOS)$
                     FOR I = ALL(BAKER)$
                     FOR E = ALL(MAXSPD)$
```

11.9.3 The Modifiers NWDS and NWDSEN

Two other less frequently used modifiers are NWDS and NWDSEN.

NWDS represents the number of words in a table. It is used in a similar fashion to NENT, but of course will always be equal to the number of entries times the number of words per entry.

If NWDS is used on the left side of an Assignment Statement, the corresponding NENT is also set (automatically) at this time.

NWDSEN is a modifier which stands for the number of words in an entry. Occasions to use this modifier are unusual.

The reason NWDS and NWDSEN are used relatively infrequently are that the language is entry oriented, so that the interpreter and translator usually read programs written in terms of entries and entry numbers and supply automatically the necessary information pertaining to number of words and number of words per entry.

11.10 Additional Remarks on the Boundaries of a Subscript

Following M consecutive FOR Statements where M is greater than or equal to 1, the M subscripts are defined, and thus legal, for one statement.

If the statement following the FOR Statement is compound, it must be enclosed in a BEGIN and END. If it is simple, it doesn't have to be so bracketed. However, one may imagine that a BEGIN and END exists around a simple statement, so that it will not hurt the following discussion if it is assumed that a BEGIN and END always exist around the statement following the FOR Statements. (In fact, the use of a BEGIN and END around a simple statement is legal.)

If the FOR Statement (or Statements) is incomplete the BEGIN and END bracketing the statement following serve only one purpose, that being the specification of the boundaries of the subscript.

Look at the series of statements:

```
          FOR I = NUMB$
     BEGIN ABLE($I$)  = 6$
          BAKER($I$) = 2$
          GOTO A1$
     END
```

This could have been written:

```
          FOR I = NUMB$
     BEGIN ABLE($I$) = 6$
          BAKER($I$) = 2$
     END
          GOTO A1$
```

since no I is used following the END.

In this example one might wonder what happens when ABLE(J) is not equal to 3. If J had been defined with an incomplete FOR Statement, it would have executed A3 and thus performed a transfer of control to A2. However, the END preceding statement A3 represents the modify and test statements on J. Thus the false branch at A1 would cause the modification and test on J and consequently a return to A1 if not all ABLE's had been tested.

Now another case may be examined.

```
FOR K = ALL(BAKER)$
     BEGIN BAKER($K$) = Ø$
          B1. IF ABLE($K$) = 1$
     END
          STOP$
```

The effect of this sequence will be to set all BAKERs to Ø up to the point where ABLE is not equal to 1. At this point, the false branch at statement B1 will cause a transfer around the END, and thus the loop will be left, and the STOP takes effect. Thus in this case the true branch is the only one which will permit the continuation of the loop, by executing the modify and test of K at the END.

Now one more problem remains. Suppose the program is in the middle of a rather long loop, one which is made up of a number of statements. For some reason, it is deemed necessary to complete the particular iteration (with the modify and test of the subscript) prior to the execution of all the statements up to the END.

For example, in the following sequence, it is desired to skip statements S4, S5, and S6 and proceed to the next iteration if ABLE(K) = Ø.

```
          FOR K = ALL(ABLE)$
     BEGIN S1$
          IF ABLE($K$) EQ Ø$
          GOTO???
          S4$
          S5$
          S6$
     END
```

Section 11.11 will discuss the mechanism for accomplishing this problem of getting to the END from the middle of the loop.

11.11 The Operator TEST

The operator TEST permits a jump to the END of a subscript from anywhere inside a compound statement for which a subscript is defined with a complete FOR Statement.
Its simplest form is illustrated in the following example.

It could not, however, be written in the following (illegal) example:

```
          FOR I = NUMB$
     BEGIN ABLE($I$) = 6$
     END
          BAKER($I$) = 4$
          GOTO A1$
```

Once the END has been reached, no more use of I is permitted.

The above example illustrates the purely "grammatical" nature of the END when used as the right bracket following the definition of a subscript with an incomplete FOR Statement.

When a subscript is defined with a complete FOR Statement, on the other hand, the END takes on additional significance.

Take the following example:

```
          FOR I = ALL(ABLE)$
     BEGIN ABLE($I$) = 6
          BAKER($I$) = 2
     END
          GOTO A1$
```

This is significantly different than the following:

```
          FOR I = ALL(ABLE)$
     BEGIN ABLE($I$) = 6$
          BAKER($I$) = 2$
          GOTO A1$
     END
```

In the latter ABLE($Ø$) will be set to 6 and BAKER($Ø$) to 2, and the GOTO will cause an immediate exit from the loop. In the former example, however, all ABLE and BAKER will be set prior to the transfer of control to A1. In other words, the END of the definition of a complete subscript stands for some actual statements, which are the modification and test of the subscript and return to the statement following the BEGIN. (In addition it serves the "grammatical" purpose of defining the end of the legal use of the subscripts.)

The fact that this END serves a dual purpose has some interesting effects on programming which must be made clear.

Examine the following example:

```
     FOR J = ALL(ABLE)$
     BEGIN A1. IF ABLE($J$) EQ 3Q
                  BAKER($J$) = 1$
     END
                A3. GOTO A2$
```

```
            FOR I = ALL(ABLE)$
        BEGIN IF BAKER($I$) EQ Ø$
            TEST$
            ABLE($I$) = 3* BAKER($I$)$
        END
```

The operator TEST as used above causes a transfer of control to the first END of a subscript defined with a complete FOR Statement.

Another example is the following:

```
               FOR I = ALL(BAKER)$
        BEGIN  FOR J = ALL(AUTOS)$
               BEGIN S1$
                     S2$
                     S3$
               S4. TEST$
                     S5$
                     S6$
               END
        END
```

This time the operator TEST will cause a transfer to the modify and test on subscript J, and therefore statements S5 and S6 will not be performed when statement S4 is executed.

If in the latter example, one had desired to modify and test I rather than J at statement S4, the format TEST I$ is available, which says skip the normal TEST transfer and go to the subscript specified.

```
Example:           FOR A = ALL(BAKER)$
           BEGIN   FOR B = ALL(DOG)$
           BEGIN   FOR C = ALL(EASY)$
           BEGIN   FOR D = PIG$
           BEGIN   S2$
              S3.  TEST$
                   S4$
              S5.  TEST B$
                   S6$
              S7.  TEST A$
                   S8$
           END
           END
           END
           END
```

At S3, control would transfer to the second end. At S5, control would be given to the third END, and at statement S7, control goes to the fourth END in the sequence. Note that no test ever goes to the first END, which ends a subscript defined with an incomplete FOR Statement.

11.12 Use of a Variable B-Factor - Variable Length Entry Tables

All examples used thus far have considered fixed length entry tables. At present, there is nothing in the language or the method of variable definition which pertains directly to tables with variable length entries. (Undoubtedly there will be someday.) Also not discussed so far has been the use of a variable B-factor in the complete FOR Statement. Although the variable B-factor can have many applications, it can be used most directly in the handling of variable length entry tables. For an example of its use and the handling of variable length entries we shall discuss a hypothetical problem.

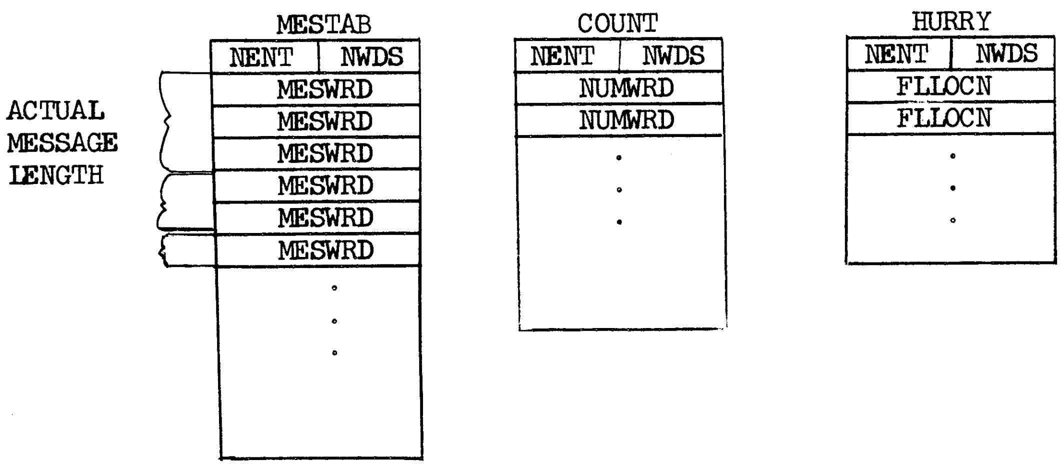
First of all, we shall consider a table of messages made up of Hollerith characters. Each message may consist of a variable number of characters, and thus each entry (message) in the table may consist of a variable number of computer words. This table will be called MESTAB. To make possible the handling of this variable length entry table it will be defined as a table with one item (only one word per entry). This will be defined as a 6 character Hollerith type item (assuming 36 bit computer words). The name of this item will be MESWRD.

Another table, called COUNT, will be defined. It also has only one item, called NUMWRD. This item is an integer which contains the number of words in the message which is placed in the same relative position within MESTAB as itself. Thus both MESTAB and COUNT are variable length tables which have equal numbers of entries at any given time. But, whereas MESTAB in fact has variable length entries, COUNT has actually a fixed entry length of one.

Lastly, there is a variable length table called HURRY. This table also has one item, called FLLOCN. This item is assumedto have the address, relative to the location of MESTAB, of a message which contains the characters "FLASH" in the first five positions of the message. There may be any number of this type of message.

Figure 3 contains a diagram of these three hypothetical tables.

FIGURE 3



The problem to be solved is the cycling through all of MESTAB, and placing the location of all FLASH messages in the table HURRY.

The solution will now be presented. (Note: The solution requires the use of the modifier BYTE, which is described in Section 13.2.)

```
  TABLE MESTAB V 1ØØØ$
BEGIN ITEM MESWRD H 6$  END

  TABLE COUNT V 5ØØ$
BEGIN ITEM NUMWRD I 6 U$  END

  TABLE HURRY V 5ØØ$
BEGIN ITEM FLLOCN I 1Ø U$  END

  FOR I = ALL(NUMWRD)$
  FOR J = Ø, NUMWRD($I$), NWDS(MESTAB)$
  FOR K = ALL(FLLOCN)$
BEGIN IF BYTE($Ø,5$)(MESWRD($J$)) EQ 5H(FLASH)$
     FLLOCN($K$) = J$
END        STOP$
```

12. SWITCHES - THE SWITCH DECLARATION AND SWITCH CALL

So far the only decision making function discussed has been the IF Statement. A frequently necessary type of function is a switch, or n-way branch, based on the value of a particular item or subscript.

In order to set up a switch in JOVIAL, one must define it with a SWITCH Declaration. The purpose of the declaration is the listing of the statement labels to which to transfer and the conditions under which the respective transfers are to be made.

The actual switch (or n-way branch) is accomplished with a SWITCH CALL. This is a particular type of GOTO Statement.

There are two types of switches, one on items and the other on subscripts.

12.1 Item Switches

12.1.1 Item SWITCH Declarations

For the following discussion, the notation will be as follows:

SNAMEi is the name of the switch.
NAMEi is the name of an item.
Vi is a particular value of the item.
Si is a statement label.

i = 1,2,...n.

The form for an item SWITCH declaration is the following:

```
SWITCH SNAME1(NAME1) =(V1=S1,V2=S2,...Vn=Sn)$
```

where NAME1 can be any type of item, V1, ...Vn can be any legitimate value of NAME1, and S1...Sn can be any statement label of the object program. Vi cannot equal Vj, but any Si can equal any Sj.

12.1.2 Item SWITCH Call

If NAME1 belongs to a table, the format of the Item SWITCH Call is: GOTO SNAME1(I)$

where I can be any legally defined subscript.

If NAME1 is an item in a one entry table or it is not a member of any table, the proper format is:

```
GOTO SNAME$
```

12.1.2.1 The Effect of the Item Switch Call

At the Switch Call Statement, all values named in the SWITCH Declaration are compared against the item (subscripted, if necessary). If any Vj agrees with the current state of the item, a transfer of control is made to the respective statement Sj. If no Vj agrees with the current value of the item, then control procedes to the statement following the SWITCH call.

Examples:

(a) For ABLE an Integer Type Item:

Declaration: `SWITCH ALPHA(ABLE)=(1=S1,5Ø=Z1A,1ØØ=ABC)$`

Call: `GOTO ALPHA($J$)$`

In this example, if the Jth ABLE =1, transfer of control is to S1, if it is 5Ø, the transfer is to Z1A, and if 1ØØ it goes to ABC. If ABLE (J) equals any other value control procedes to the statement following the call.

(b) For COL a Hollerith Type Item with 2 Hollerith characters:

Declaration:

```
SWITCH BETA(COL)=(2H($))=CDE,2H(OQ)=SØ4,2H(PQ)=SST,2H(1H)=PQR)$
```

Call: `GOTO BETA($K$)$`

(c) For TAPSTT a Status Type Item not in a table:

```
Declaration:  SWITCH GAMMA(TAPSTT)= (READY,=GO, NOTRED=STOP,
                                    EOF=Z3A)$
```

Call: `GOTO GAMMA$`

(Note the currently existent inconsistency of format. The status constant in this case does not have the letter "V" and parentheses, as it usually does).

12.2 Subscript Switches

12.2.1 Subscript SWITCH Declarations

Subscripts are assumed to take on only positive, or zero, integer values for switches. In the subscript SWITCH Declaration, all possible values of the particular subscript must be accounted for.

The form of the SWITCH Declaration is:

```
SWITCH SNAME2 = (S1,S2,S3,...Sn)
```

If it is desired to have control procede to the next statement for certain values of the subscript, this is signified by consecutive commas, such as:

```
SWITCH SNAME3 =  ( ,S1,,S2,S3,S4,,,S5)
```

In this example, the values ∅, 2,6, and 7 will cause a transfer to the statement following the SWITCH Call.

12.2.2 Subscript SWITCH Calls

The form for this type of call is:

GOTO SNAME2(I)$, where I is any legal subscript.

Example:

```
Declaration:  SWITCH EPSLON=(S∅A,,ZEBO,,AMOS)$

Call:         GOTO EPSLON ($J$)
```

In this example, if J=1 or 3, control proceeds to the statement following the call. If J equals ∅, control goes to S∅A if 2, control goes to ZEBO, and 4 to AMOS.

Note: It is the programmer's responsibility to make certain that J is not greater than 4, or, in general, n-1 where n is the number of values accounted for in the declaration.

13. USING PARTS OF ITEMS - THE MODIFIERS BIT, BYTE, CHAR, MANT

Access to parts of items is permitted in JOVIAL. Certain modifiers preceding an item name will allow one to get the information and use it, or, when used as the left term of an Assignment Statement, will permit the setting of just the specified part of the item. One general rule prevails whenever these modifiers are used. The information used or set in this fashion is treated as if it were an unsigned Integer Type Item, even if the part of the item being used includes a sign or non-integer information (e.g., bits to the right of the binary point).

13.1 The Modifier BIT

The modifier BIT allows access to specified bits of an item. The starting bit and number of bits are specified by subscripts enclosed in the brackets "($" and "$)" following the modifier.

Examples:

```
BIT($ø,2$)(ABLE)
BIT($I,3$)(BAKER)
BIT($3,J$)(CHARLY)
BIT($K,L$)(DOG)
```

If it is desired to specify just one bit, two alternative notations are available. In this case, the number of bits does not have to be specified.

Examples: BIT($2,1$)(ABLE) is equivalent to BIT(2)(ABLE).
BIT($ø,1$)(DOG) is equivalent to BIT($ø$)(DOG).

BIT($ø$) has special significance. For signed items, it represents the sign bit. For unsigned items, it is the first magnitude bit.

Example of programming Using the modifier BIT:

```
                FOR I = ø,1,7$
                FOR J = ALL(BAKER)$

BEGIN IF BIT($I$)(ABLE($J$))EQ 1$

                BIT($J,2$)(BAKER($I$)) = 3$

END
```

13.2 The Modifier BYTE

The operator BIT permits access to the bits of any item. The operator BYTE permits access to any number of 6 bit Hollerith characters within a Hollerith Type Item.

```
Examples:  BYTE($2$)(COL)
           BYTE($J$)(COL)
           BYTE(Ø,3)(COL)
           BYTE(I,2)(COL)
           BYTE(3,K)(COL)
           BYTE(A,B)(COL)
```

13.3 The Modifier CHAR

The characteristic of a floating point number may be referred to with the modifier CHAR.

Example: CHAR(MAXSPD)

13.4 The Modifier MANT

The mantissa of a floating point item may be referred to with this modifier. This will include only the magnitude bits. The sign is not referenced with this modifier.

Example: MANT (MAXSPD)

14. REFERENCING COMPLETE ENTRIES - THE MODIFIER ENT

A limited amount of manipulation of complete entries of tables is available with the modifier ENT. An entry can be set to Ø (or some other constant, which isn't advisable) or one entry can be set to another entry.

When an entry is set to zero, all words in the entry are set to zero. When one entry is set to another, the number of words in each must be equal.

The format for the use of this modifier assumes that one wishes to refer to "the entry containing some ITEM(I)", where I is any subscript.

Thus the format is:

```
      ENT(ITEM($J$))
OR ALTERNATIVELY, ENT(TABLE($J$))
```

Examples of use:

```
      ENT(CYL($J$)) = Ø$
      ENT(HP($K$)) = ENT(HP($L$))$
```

15. COMMENT STATEMENTS - THE COMM DECLARATION

Although JOVIAL has a fair amount of readability inherent in its structure, occassions arise when additional text is desirable to shed light on the statements of the object program. Statements for this purpose, called Comment Statements, are preceded with the declaration COMM. Following this declarator, any set of words, numbers and characters may be used, with two restrictions.

1) The first character of the comment must be a letter or number.

2) Within the body of the comment, the symbol "$" can be used only in one of the combinations "($" or "$)".

The symbol "$" is used to mark the end of a comment statement.

Examples:

```
COMM THIS IS A */.)( - PROGRAM $

COMM IS ABLE ($I$)  OVER 16$
```

16. CLOSED SUBROUTINES - PROCEDURES AND "CLOSED" ROUTINES

The discussion so far has described the various properties of JOVIAL as they would be used in programming the "main" program, that is, no mention has been made of the technique for writing closed subroutines. Since writing and using subroutines in JOVIAL require some special devices and knowledge of certain rules, this chapter will be devoted to a discussion of these.

16.1 Definitions

This section will define certain terminology which is peculiar to the writing and using closed subroutines.

a) Procedure - A closed subroutine in JOVIAL is in general called a procedure. (One exception is a "Closed" routine, described in Section 16.7.) A procedure has one entrance and one exit point. It cannot be entered via a direct flow of the program.

b) Procedure Declaration - A procedure declaration is a statement which declares a set of statements to be a procedure.

c) Procedure Call - A procedure call is the link from the main program to a procedure. It is the only place from which a procedure may be entered.

d) Input Parameter - An input parameter is an arithmetic expression specified in the procedure call which represents a value on which the procedure is to operate.

e) Output Parameter - An output parameter is an item specified in the procedure call which is to contain an output of the procedure.

f) Dummy Input Parameter - A dummy input parameter is an item specified in the procedure declaration which represents a value to be used by the procedure as an input parameter.

g) Dummy Output Parameter - A dummy output parameter is an item specified in the procedure declaration which represents a value to be set by the procedure as an output parameter.

h) No Output Procedure - A no output procedure is one which uses input parameters only. If it sets any items or tables, they are always the same ones, since no output parameter may be specified in the procedure call.

i) Multiple Output Procedures - A multiple output procedure is one which sets one or more output parameters.

j) One Output Procedures - A one output procedure is one in which no output parameters are specified but which produces as its output a single value which is to be used immediately for further calculation by the main program.

16.2 Formats

This section describes the format for the various procedure declarations and calls. In all cases except the one output procedure call, the format is a complete statement.

Note: The notation is as follows:

ID - The name of the procedure. (This must be 2 to 6 alphanumeric characters, of which the first must be a letter.)

Pi - Input Parameter - This may be any arithmetic expression.

Oi - Output Parameter - This must be an item.

Di - Dummy Input Parameter - This must be an item.

Qi - Dummy Output Parameter - This must be an item.

a) No Output Procedure Declaration:

PROC ID (D_1,D_2...D_m) \$ $\{m \geq 1\}$

b) No Output Procedure Call:

ID (P_1,P_2...P_m) \$ $\{m \geq 1\}$

c) Multiple Output Procedure Declaration:

PROC ID (D_1,D_2,..D_m = Q_1,Q_2,...Q_n)\$ $\{m \geq 1, n \geq 1\}$

d) Multiple Output Procedure Call:

ID(P_1,P_2,...P_m=O_1,O_2,...O_n)\$ $\{m \geq 1, n \geq 1\}$

e) One Output Procedure Declaration:

PROC ID(D_1,D_2,...D_m)\$ $\{m \geq 1\}$

f) One Output Procedure Call:

ID(P_1,P_2,...P_m) $\{m \geq 1\}$

Note that (f) is never a complete statement. It is always part of an arithmetic expression. (It may be the only part.)

16.3 General Rules for Procedures

a) All dummy input and output parameters must be defined with Item Declaration Statements immediately following the Procedure Declaration. In addition, all items and tables which are used solely by the procedure (those which are not defined in the Communication Pool or the main program) must be defined with Item and Table Declaration Statements following the Procedure Declaration. The Procedure Declaration plus the list of variable declarations is called the Procedure Heading. All items and tables defined in the Procedure Heading may have the same name as items and tables defined outside the procedure.

b) Items and tables used by a procedure which are defined outside of the procedure must not be defined in the Procedure Heading.

c) Immediately following the Procedure Heading (preceding the first dynamic statement of the procedure) the bracket "BEGIN" must appear. Immediately following the last statement of the procedure, the bracket "END" must appear. The latter bracket serves as the unique exit from the procedure.

d) A Procedure Declaration may not appear within a procedure, or within any pair of BEGIN, END brackets.

e) One or more Procedure Calls (of other procedures) may appear within a procedure. At present, only four "levels" of calls may exist.

f) Statement Labels within procedures are considered to be unique for the procedure. Therefore statement labels used within a procedure may duplicate those in the main program or in other procedures. Procedure names, however, can not be duplicated.

g) All subscripts referred to by a procedure must be defined by FOR Statements within the procedure. However, the current value of a subscript used outside the procedure may be transmitted to a procedure via an input parameter. Likewise, the value of a subscript set within a procedure may be transmitted to the outside of the procedure with an output parameter.

h) There must be a one-one correspondence of input and output parameters to dummy input and output parameters, respectively. In other words, there must be one parameter for each dummy parameter. The respective positions of the parameters must be identical to the positions of the dummy parameters within the Procedure Declaration. Also the type and scaling of each parameter must agree exactly with the type and scaling of its respective dummy parameter as defined in the procedure heading.

i) Multiple Output and No Output Procedure Calls cannot be used as input parameters.

j) The exit of a Multiple Output or No Output Procedure is to the statement following the Procedure Call.

16.4 Special Rules for One-Output Procedures

Although all rules mentioned in Section 16.3 hold for One-Output Procedures as well as others, the following rules apply to One-Output Procedures only.

a) In the heading of a One-Output Procedure Declaration an Item Declaration Statement for an item **with the same name as the procedure** must appear. This is called the Output Item.

b) The Output Item must be set prior to exiting from a procedure. This item must contain the value which is the result of the procedure's operation.

16.5 The RETURN Statement

In (c) of Section 16.3 it was stated that the bracket "END" which marks the physical end of a procedure also serves as the dynamic exit point of the procedure. Therefore if a procedure consists of a series of statements which always dynamically reach the point of the END, no additional operators are necessary. However, procedures which do not have this required linearity need some means of reaching the dynamic and physical END of the procedure. The means provided for this is the operator "RETURN" which is used in a RETURN statement. The RETURN Statement <u>always</u> has the format;

```
RETURN$
```

The function of this statement is to perform a transfer of control to the "END"of the procedure, which in turn causes an exit from the procedure.

16.6 Examples:

In Example (2) of Section 8.2.1.1, a program to compute the square root of an item was written. To illustrate as many points from this chapter as possible a number of variations on the theme of this square root will be given.

16.6.1 One Output Procedure with One Input Parameter

One Output Procedures are usually thought of in terms of mathematical functions like square root, sin, arctan, etc. This example will illustrate the writing of a simple square root routine which computes the square root of any posivite floating point item. Then the use of this procedure will be illustrated in a program which computes the hypotenuse of 100 triangles.

How to write the procedure:

```
        PROC SQRT(SQUAR)$
        ITEM SQRT  F$ COMM THIS IS THE OUTPUT$
        ITEM SQUAR F$ COMM THIS IS THE INPUT$

BEGIN SQRT = .5* SQUAR$
        A2. IF ABS(SQRT(*2*) - SQUAR) GQ .ØØØ1$
        BEGIN SQRT = .5*(SQRT+SQUAR/SQRT)$
           GOTO A2$  END
END
```

How to use the procedure in a program:

```
  START$
  TABLE TRIANG R 1ØØ$
 BEGIN ITEM LSIDE F$
       ITEM RSIDE F$
       ITEM HYPOT F$  END
  FOR I = ALL(HYPOT)$
  HYPOT($I$) = SQRT(LSIDE($I$)(*2*)+RSIDE($I$)(*2*))$
  TERM$
```

Note that the input parameter is one floating arithmetic expression and therefore meets the requirement of the one dummy input parameter which has been defined as a floating point item.

16.6.2 <u>One Output Procedure with two Input Pararmeters</u>

The procedure in section 16.6.1 always computed the square root so that the difference between its square and the true square was less than ØØØ1. We shall now write and use the same routine, except that this time we shall permit the difference to be specified in the call of the procedure rather than the constant .ØØØ1.

```
          PROC VSQRT(SQUAR,EPSLON)$
               ITEM VSQRT F$
               ITEM SQUAR F$
               ITEM EPSLON F$
          BEGIN  VSQRT = .5* SQUAR$
               A2. IF ABS(SQRT(*2*) - SQUAR) GQ EPSLON$
            BEGIN VSQRT = .5*(VSQRT + SQUAR/VSQRT)$
                  GOTO A2$ END  END
```

Computing the 100 hypotenuses:

```
          START $
          TABLE TRIANG R 1ØØ$
   BEGIN  ITEM LSIDE F$ ITEM RSIDE F$
          ITEM HPOT F$ ITEM DELTA F$   END
          FOR I = ALL(LSIDE)$
   HYPOT($I$) = SQRT(LSIDE($I$)(*2*)+RSIDE($I$)(*2*),DELTA($I$))$
       TERM$
```

16.6.3 Multiple Output Procedure

Until now, we have been assuming that the input parameter is always positive, and therefore the square root could be computed. Also, using the procedures so far written we could use the square root computed in any fashion as long as it were part of an arithmetic expression. Now we shall write and use a procedure which, using a variable EPSLON, tests for a negative square, and if found sets an item to 1. In addition, it will always set an item to the computed square root. This is an example of a Multiple Output Procedure.

```
          PROC SSQURT(SQUAR, EPSLON = ERR, SQRT)$
          ITEM SQUAR F$
          ITEM EPSLON F$
          ITEM ERR I 1 U$
          ITEM SQRT F$
     BEGIN ERR = ∅$
          IF SIGN(SQUAR) EQ 1$
          BEGIN ERR = 1$ RETURN$ END
          SQRT = .5*SQUAR$
          A2. IF ABS(SQRT(*2*) - SQUAR) GQ EPSLON$
          BEGIN SQRT = .5*(SQRT + SQUAR/SQRT)$ GOTO A2$ END
          END
```

Now, using the procedure in a program:

```
          START$
          TABLE TRIANG R 1∅∅$
          BEGIN ITEM LSIDE F$ ITEM RSIDE F$ ITEM HYPOT F$
                ITEM ALARM I 1 U$  END
          ITEM DELTA F$
          DELTA = .∅∅5$
          FOR I = ALL(TRIANG)$
          SSQRT(LSIDE($I$)(*2*)+RSIDE($I$)(*2*),DELTA = ALARM ($I$),
                                                    HYPOT($I$))$
          TERM$
```

16.6.4 No Output Procedure

No Output Procedures always set a compool or main program defined table or item. The following example will illustrate the writing of a procedure which always sets the compool defined floating item SQROOT to the square root of the input parameter and the use of it to set all HYPOT's as in the previous examples.

Writing the procedure:

```
          PROC SQRR (SQUAR)$
            ITEM SQUAR F$
          BEGIN
            SQROOT = .5*SQUAR$
          A2. IF ABS (SQROOT(*2*) - SQUAR) GQ .∅∅∅1$
            BEGIN SQROOT = .5*(SQROOT+SQUAR/SQROOT)$
                  GOTO A2$ END  END
```

(Note that this type of procedure looks very similar to a One Output Procedure, except that there is no item defined with the same name as the name of the procedure.)

Using the procedure:

```
            START$
            TABLE TRIANG R 1ØØ$
      BEGIN ITEM LSIDE F$ ITEM RSIDE F$
            ITEM HYPOT F$  END

            FOR I = ALL(HYPOT)$

      BEGIN SQRR(LSIDE($I$)(*2*) + RSIDE($I$)(*2*))$
            HYPOT($I$) = SQROOT$  END  TERM$
```

16.7 "Closed Routines - The CLOSE Declaration

A procedure is the commonest form of closed subroutine used in JOVIAL. One other form of closed subroutine exists. It is called a "Closed" Routine. It has many different properties than a procedure.

Properties of "Closed" Routines are described in the following list:

(a) The CLOSE Declaration is used to head a "Closed" Routine. The format of the "Closed" Routine is the following:

```
        CLOSE ID$
     BEGIN S1$ S2$ ...Sm$ END
```

where the Si's are the statements of the routine.

(b) A "Closed" Routine has no input or output parameters.

(c) A "Closed" Routine has no special Item or Table Declaration Statements within its body or heading.

(d) A "Closed" Routine may be included, and usually is, within the brackets BEGIN and END. In other words, it may be contained within a procedure, another "Closed" Routine, or a compound statement.

(e) Like a procedure, the bracket BEGIN following the CLOSE Declaration is the unique entry point of the "Closed" Routine. Also, the bracket END represents the unique exit from the routine. The RETURN Statement may be used within a "Closed" Routine.

(f) The entry to a"Closed" Routine is made with a GOTO Statement, not a special Call Statement.

Example: GOTO ID$

(g) When a "Closed" Routine is enclosed within the boundaries of definition of a subscript, the definition of the subscript holds for the body of the "Closed" Routine. If the subscript is modified within the "Closed" Routine, the modified value will exist upon exit from the routine.

16.7.1 The following program illustrates the writing and use of a "Closed" Routine. The use of the "Closed" Routine in this case creates a rather inefficient program. The example is for illustrative purposes only.

```
                START$
                TABLE TALL V 5Ø$
          BEGIN ITEM ABLE F$
                ITEM BAKER F$
          END   ITEM ALARM I 1 U$
                FOR I = NENT(ABLE)-1,-1,Ø$
          BEGIN IF ABLE($I$) EQ Ø.$
            BEGIN GOTO SUBT$
            A1. IF ALARM EQ 1$
                STOP$ TEST$
            END IF BAKER($I$) = 1.Ø$
            BEGIN GOTO SUBT$
                   GOTO A1$ END TEST$
CLOSE SUBT$ BEGIN ALARM = Ø$
                      IF NENT(ABLE) EQ 1$
             BEGIN ALARM = 1$ RETURN$
             END ABLE($I-1$) = ABLE($I-1$) + 1.6$
             END
          END STOP$
            TERM$
```

17. STATEMENT, CARD AND DECK FORMAT

JOVIAL is, in a fashion, a programming language which approaches the English, or program specification language. Therefore, an attempt has been made to keep the structural rules as relaxed as possible. Programming and computers being what they are, however, complete laissez faire on a punched card is somewhat beyond our grasp, so that certain rules must be followed.

Experience on the part of the individual JOVIAL programmer will probably be the best teacher in the matter of statement, card, and deck construction. At best, a list of general rules can be given here. The remarks pertaining to the deck construction are reasonably accurate, but remarks pertaining to statement and card format are probably incomplete.

17.1 Statement Format

(1) Single terms, such as variables, constants, operators, etc., which consist of consecutive strings of letters or numbers, must not have any spaces between the first and last character, and they cannot be broken onto two consecutive cards.

(2) Where two consecutive entities of the type mentioned in (1) appear, they must be separated by at least one space. (e.g. SWITCH ALPHA ...)

(3) It is not necessary (although it might sometimes be desirable) to separate special characters (non-alphanumeric) from entities mentioned in (1) and (under some circumstances) from each other.

Example: AB = 3*(BC-CD)/(EF+GH)$

In this statement, no spaces need appear.

(4) Wherever one space is required, or permitted, any number of spaces may exist.

(5) No implied multiplication is allowed. The expression (AB+BC)(CD+EF) is illegal. It should be written (AB + BC) * (CD + EF).

(6) For Hollerith Type Constants, the number of characters within the parentheses must agree precisely with the number stated outside the parentheses. A blank is a legal Hollerith Character and should be included in the count.

17.2 Card Format

(1) Coding can begin at any column on the card.

(2) Coding can stop at any column on the card, but must not extend beyond column 66.

(3) More than one statement can appear on one card.

Example: The following "cards"

```
FOR I=NUMB$ BEGIN XY($I$) = 6$ AB($I$) = Ø$
    OPQ($I$) = I $ END
```

are equivalent to:

```
    FOR I = NUMB$
BEGIN
XY($I$) = 6$
AB($I$) = Ø$
OPQ($I$) = I$
END
```

(4) A statement may continue for as many cards as necessary. The end of any one card, however, cannot be the middle of a single entity such as a variable name, etc.

17.3 Deck Format

(1) The first card in any deck must be a card beginning with the declarator, START. Following this declarator, there can be no other JOVIAL Statement on the same card. Any remarks, preferably program name, date, etc., should be added since this card is logged on the printer when recognized.

(2) All Item and Table Declaration Statements except those for procedures must follow the START card and precede the first dynamic JOVIAL instruction.

(3) All procedure variable declarations must precede the first procedure dynamic instructions.

(4) The last card in the deck must have the operator TERM followed by the symbol "$" or a statement label followed by the symbol "$". If the TERM is not followed by a statement label, the first operating instruction is considered to be the first JOVIAL dynamic instruction. If it has a label, then the statement having this label will be considered to be the first dynamic instruction. Other JOVIAL Statements may precede the operator TERM on the same card, but none can follow.

(5) Procedure declarations may not appear where they will be operated in the normal flow of the program.

The following two examples are <u>illegal</u>.

```
(a)  ABLE = BAKER$
     CHARLY = DOG$
     PROC ID...
(b)  IF ABLE EQ 3$
     BAKER = 7$
     PROC ID...
```

(6) Switch Declarations may not appear immediately following an IF Statement.

The following example is illegal:

```
IF ABLE EQ 3$
SWITCH ID...
```

18. SUMMARY

This section contains a concise summary of the language. The detailed descriptions of the various facets of the language have been omitted in an effort to describe the main components.

I Definitions:

t is a table.
v is an item or an item with a modifier.
o is the output of a one output procedure.
k is a constant.
s is a subscript.
A is an arithmetic operator (+,-,*,/,(*,*),ABS).
R is a relational operator (EQ,NQ,LS,GR,LQ,GQ).
L is a logical operator (AND,OR,NOT)
E is an arithmetic expression (a concatenation of v's, k's, o's connected by A's).
C is a condition (Ei R Ej)
B is a logical expression (a concatenation of C's connected by L's)
N is a string of 2 to 6 alphanumeric characters beginning with a letter.
Σ is a statement.
l is a statement label.

II Statements:

A) Item Declarations:

```
ITEM N F$            (Floating)
ITEM N I #BITS S/U$ (Integer)
ITEM N H #CHARACTERS$(Hollerith)
ITEM N A #BITS S/U ACCURACY$(Mixed)
ITEM N S ST1 ST2...$ (Status)
```

B) Item Parameter Declarations

```
ITEM N... P k$
```

C) Table Declarations

```
TABLE N R/V #ENTRIES$
BEGIN ITEM N...$
      ITEM N...$
          .
          .
          .
 END
```

D) Table of Constants Declarations

```
   TABLE N R #ENTRIES$
BEGIN ITEM N...$  BEGIN k k...$  END
      ITEM N...$  ...
END
```

E) Like Table Declarations

```
TABLE N  ... L$
```

F) Assignment

```
v = E$
s = E$
```

G) Unconditional Transfer of Control

```
GOTO 1$
```

H) Conditional Transfer of Control

```
IF  B$
```

In the event B is true, the next statement in sequence will be executed. If B is false, the next (simple or compound) statement will be skipped.

I) Unconditional Halts

```
   STOP$
or STOP 1$
```

J) Complete FOR Statements

```
FOR s = E, k, E$
FOR s = E, v, E$
FOR s = E, s, E$
FOR s = ALL(v)$
FOR s = ALL(t)$
```

K) Incomplete FOR Statements

FOR s = E$
In J and K, the subscript is defined for the next statement only.

L) Unconditional Transfer to the Boundary of a Loop

TEST$
or, TEST s$

M) Subscript Switch Declaration

SWITCH N = (1,1,...)$, where some 1 positions may be omitted.

N) Subscript Switch Call

GOTO N(s)$

O) Item Switch Declaration

SWITCH N(v) = (k = 1, k = 1 ...)$
(v may not have a modifier)

P) Item Switch Call

GOTO N$
or, GOTO N(s)$

Q) One Output Procedure Declaration

PROC N(v,v,...)$ (v may not have a modifier)

R) Multiple Output Procedure Declaration

PROC N(v,v,v... = v,v,v...)$(v may not have a modifier)

S) Multiple Output Procedure Call

N(α,α,... α = v, v, v...)$(v may not have a modifier)
where α can be v, o, s, or k.

T) No Output Procedure Declaration

PROC N(v,v,...)$(v may not have a modifier)

U) No Output Procedure Call

N (α,α,...)$

Distribution

SDC (Lodi)

All SACCS Programming Group (1 ea.)
All CUSS Project
B. H. Bragen
W. R. Goodwin
L. Ngou
H. R. Patton
A. M. Rosenberg
V. S. Thurlow

SDC (Santa Monica)

E. Book
R. Bosak
H. Bratman
W. Fitzgerald (5)
E. S. Gordon
D. E. Henley
E. Jacobs
H. A. Kinslow
J. L. Koory
R. E. Olsen
C. J. Mosmann (5)
R. Schaub
G. Dobbs (5)

SDC (Omaha)

S. Dorresteyn
G. J. Keckhut
S. Levy
J. C. Rea

IEC

c/o Ruth Prabetz for:

IEC Standard Distribution

RAND

c/o Margaret Anderson for:

C. Baker
M. Bernstein
I. Greenwald

BTL

c/o Mrs. R. M. Riley for:

G. Clement
C. Sherrerd (2)

IBM

c/o L. F. Witte for:

R. Washburne
P. Metzger

IBM (At IEC)

B. Lorber

MITRE

c/o Mrs. Jean Claflin for:

J. Burrows (3)
J. Porter
E. Bensley

RADC

Commander, Rome Air Development Center
Attn: Mr. J. Widrewitz (RCCS)
Rome, New York (2 copies)

SAC

c/o Margaret Cameron for:

Captain Phythyan (3)
DOCOA
Hq. SAC
Offutt Air Force Base
Nebraska

JIS:lb