TM-555/005/00

Grammar and Lexicon for Basic JOVIAL

10 May 1966

TECHNICAL MEMORANDUM

(TM Series)

This document was produced by SDC in performance of U. S. Government Contracts

Grammar and Lexicon for Basic JOVIAL

by

Millard H. Perstein

SYSTEM DEVELOPMENT CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA
90406

10 May 1966

SDC

SDC Memo M-14658 of August 20, 1965, signed by all Division Managers, directed standardization activity with respect to JOVIAL. This document is the standard specification for the JOVIAL subset. "Basic JOVIAL" has been selected as a more appropriate name for this standard subset than "JS." This document, together with TM-555/002/04 mod B (to be published shortly), completes the implementation of Action Item 1 of M-14658.

The policy regarding JOVIAL compilers as they relate to the specifications for Basic JOVIAL and for J3 is stated in the preface, page 3.

The substantive ways in which this document differs from TM-1682/003/00 are summarized below:

- Lowercase Greek letters are used to indicate certain system-dependent numeric quantities that must be specified for each implementation (Table I, page 11).
- The unitary and explicit nature of the specification is stated (Section 1.5, page 12).
- Limitation on the sizes of *constants* is relaxed (Section 2.6, page 17).
- *Hollerith* may include system-dependent characters (Section 2.6.1, page 17).
- *Transmission:code* is eliminated (Section 2.6.1, page 17).
- *Fixed* intermediate results are permitted (Section 2.6.1, page 17; Section 3.3.4, pages 32-34).
- Computer representation of *constants* and *variables* is specified (Section 2.6.4, pages 20-21).
- Side effects of *functions* are not permitted to affect the value of a *formula* (Section 3.3.1, page 29; Section 3.5.1, page 38).
- Absolute value indication (/ /) and exponentiation indication ** are permitted (Section 3.3.3, page 30).
- Precision of arithmetic comparisons is specified (Section 3.3.7, page 35).
- Comparison of *literal:formulas* is restricted (Section 3.3.8, page 35).
- Some details of evaluating *boolean:formulas* are specified (Section 3.3.8, page 36).
- Order of assignment of *procedure:parameters* is specified (Section 3.5.5, page 41; Section 5.5, page 70).

(continued overleaf)

- Anomalous interpretation of the second *formula* in a *for:clause* is eliminated (Section 3.7.4, page 44).
- *Test:statement*, in conjunction with a *one:factor:for:clause*, terminates execution of the "loop" (Section 3.7.7, pages 47-49).
- The arbitrary nature of "word size" and other system-dependent parameters is explained (Section 4.4.3, pages 53 ff).
- New factors must be considered in allocating space by means of *overlays* (Section 4.4.3, pages 53 ff).
- New specifications with regard to control words and packing in *tables* (Section 4.5.5, page 57).
- No nesting of *close:declarations* (Section 5.4, page 68).
- Built-in *function* REM and *procedure* REMQUO are specified (Section 5.7, page 7).
- A *program* may be given an *origin* or compiled as a closed subroutine (Section 6.1, page 73).
- *Function:names* need not be declared prior to use (Section 6.4, page 75).
- There is a discussion of the sequence of evaluation in various *formulas* (Sections 6.5-6.5.3, pages 76-80).

(page 2 blank)

Grammar and Lexicon for Basic JOVIAL

by

Millard H. Perstein

A B S T R A C T

This is the reference manual for the Basic JOVIAL version of the JOVIAL language. Basic JOVIAL is a subset of J3, which is the full language. This manual defines the full syntax and semantics of Basic JOVIAL without drawing fine distinctions between the two aspects.

(page 4 blank)

PREFACE

This manual is a complete specification of the Basic JOVIAL version of the JOVIAL language. The author believes that both users and implementers must use the same document in deciding questions of legality and meaning for a programming language. This document is intended to be that final authority for Basic JOVIAL. If it fails in meeting this goal in any particular, the author would appreciate prompt notification so that timely corrections may be issued.

This reference manual is not intended primarily as a textbook on the language. However, it may well be that no textbook will be available for some time; consequently, a style has been adopted and expository material and examples included to make the specifications meaningful to programmers unfamiliar with JOVIAL.

When specific questions arise, it is suggested that the index (and glossary) be consulted. The index has been constructed so that many questions are answered directly, for those already familiar with the language, without the need for reading the referenced sections.

There are gaps in the sequence of section and table numbers because this manual is based on The JOVIAL (J3) Grammar and Lexicon, TM-555/002, the specification for the J3 language. The numbers of sections for which there are no corresponding sections in this document are simply omitted. The numbers that do appear match the corresponding discussions in TM-555/002.

With the publication of this manual, Grammar and Lexicon for Basic JOVIAL (TM-555/005), the policy with regard to any new JOVIAL compiler is that it must implement Basic JOVIAL as defined herein. If the new compiler provides capabilities included in J3, it must implement them in accordance with The JOVIAL (J3) Grammar and Lexicon (TM-555/002). It may implement features not included in the specifications of JOVIAL and not incompatible with the specifications of JOVIAL. The new compiler must be fully documented with respect to those features of JOVIAL described as system-dependent and with respect to how it differs from J3, from Basic JOVIAL, or both.

BLANK PAGE

CONTENTS

CONTENTS (continued)

(page 8 blank)

CONTENTS (continued)

BLANK PAGE

CHAPTER 1. INTRODUCTION

1.1 PURPOSE OF THIS DOCUMENT

This is the reference manual for Basic JOVIAL, a standard subset of JOVIAL. For the implementer or compiler writer who prefers to work from syntactic charts, or specifications in a syntax-description language, translation of this manual to one of these forms is necessary. For the relatively inexperienced student, tutorial documents must be derived from this manual.

This manual goes into some detail concerning aspects of JOVIAL that were not mentioned in earlier versions of JOVIAL language specifications. These details are, to varying degrees, machine-dependent or system-dependent. However, it is particularly in these previously unspecified details that implementations of the language differ. The differences are often undocumented or documented in series of notes that have a propensity for getting lost.

Adequate documentation for any implementation of JOVIAL must include specification of the values represented here by lowercase Greek letters; explanations of any exceptions, deviations, or additions to the forms and meanings (syntax and semantics) specified herein; and further specification of those features described herein as compiler-dependent or system-dependent.

There is an alternative: if the new compiler handles a language that more closely approximates J3 than Basic JOVIAL, it may be documented by following the steps of the previous paragraph, but with respect to the J3 manual rather than the Basic JOVIAL manual.

1.2 THE DESCRIPTIVE METALANGUAGE FOR JOVIAL

The language of this document consists of the following elements:

1. JOVIAL symbols
2. the English language
3. lowercase Greek letters
4. other words and phrases
5. numbers
6. punctuation and other special symbols
7. arrangement on the page
8. diagrams, tables, and figures

Certain elements of JOVIAL look just like the punctuation used with English, for instance the comma and the period that are parts of this sentence. No attempt is made to distinguish these classes formally, but context should provide the required distinctions.

The "other words and phrases" are distinguished from both JOVIAL words and English words by being typed in italics. An example, to be defined later, is "*letter*." All such "other words" are spelled like English words and have similar, but not identical, meanings. For instance, "*letter*" refers to an element of the JOVIAL alphabet, whereas "letter" refers to an element of the English alphabet; a "*space*" is an element of JOVIAL, whereas a "space" is only a device to separate parts of this description.

Defining sentences, formulas, and lists use capital letters and numerals, typed in elite, as specific instances of themselves in JOVIAL, and "other words and phrases," typed in italics, as representative members of classes of JOVIAL elements. English words (in elite and lower case with normal capitalization), such as "followed by" and "or," are used at times to indicate such as things as order and alternatives. If punctuation is present:

9. In sentences, it is English and part of the sentence.
10. In formulas, it is JOVIAL and part of the expression.

Punctuation may appear in either elite or italics, with no meaning attached to the difference.

Throughout the document there are lists of alternative defining formulas and of examples. Some elements of lists require more than one line. In order to distinguish the elements unequivocally, they are numbered at the left as in the previous paragraph. The number and its following period is never a part of the formula or example.

Italic words or phrases written one after another, with one or more intervening spaces, indicate concatenation. For instance, the formula:

11. *letter letter*

means the same as *letter* followed by *letter*. In order to join such words together to form descriptive names for classes, a colon is placed between the words. If such a phrase begins on one line and continues on the next, the colon is repeated. For instance--*formal:input:parameter:*
:list. Such a phrase is never broken within a word. Here are four examples of phrases (to be defined later) naming classes of JOVIAL structures:

12. *formal:input:parameter:list*
13. *formal:output:parameter:list*
14. *actual:input:parameter:list*
15. *actual:output:parameter:list*

Since spaces do not indicate *spaces*, the special symbol θ is used in formulas to indicate that one or more *spaces* are permitted. Depending on the particular realization of the formula, a *space* may be required, as explained in Section 2.7.

There are italic words or phrases, not explicitly defined, used in describing JOVIAL structures. These phrases are derived by breaking up or putting together other phrases. The meanings are obvious. For instance, it should be clear that the following four examples are four of the six possible combinations of the classes named in examples 12, 13, 14, and 15, taken two at a time:

16. *input:parameter:list*
17. *output:parameter:list*
18. *formal:parameter:list*
19. *actual:parameter:list*

There are many numbers that must be specified in describing Basic JOVIAL as it is implemented by a particular compiler. Some of these, such as size of a word, relate directly to the semantics of the language, while others are concerned with capacities in the compiler. Some of these numeric values are mentioned in general terms elsewhere in this manual. The Greek letters in Table I are used to represent these values.

Table I. Greek Letters for System-Dependent Parameters

α	maximum bits in a *literal:item*
β	maximum bytes in a *literal:item*
γ	maximum words in a *literal:item*
δ	bits per word
λ	bits per signicand in *floating:item*
τ	largest integer in $(\pi+2)/3$
υ	bits per exrad in *floating:item*
ψ	bytes per word
ω	bits per byte
π	maximum bits in an *integer:item*
η	the number of bits in a *floating:item*
ν	the number of bits in the basic addressable unit

1.3 ONE-DIMENSIONAL NATURE OF A *PROGRAM*

Regardless of the forms used for coding, the input medium, or the arrangement of the coding on that medium, the language definition considers a JOVIAL *program* to be a continuous stream of JOVIAL *signs*.

1.4 SYNTAX AND SEMANTICS--ILLEGAL, UNDEFINED, UNGRAMMATICAL, COMPILER-DEPENDENT

This manual makes no great distinction between syntax and semantics. It gives complete specifications, however, for writing legitimate Basic JOVIAL *programs*. In those instances when structure or meaning is described as system-dependent or compiler-dependent, the user must consult other documentation (or write the additional documentation if he is building the compiler) to learn of further restrictions. This other documentation gives numeric values for the Greek letters in Table I, explains in detail how the particular compiler deviates from these specifications, and lists and explains all error messages that may be generated by the compiler.

For a *program* to be legitimate it must be meaningfully structured in accordance with the specifications in this manual. If the *program* or any part of it fails to meet this requirement, it is of small concern whether it be called illegal, undefined, or ungrammatical.

It often happens that compilers do not reject certain illegal or undefined structures, but compile them instead, giving results that the programmer considers appropriate. It is recommended that programmers avoid exploiting these quirks, since there is no guarantee that a new version of the compiler will exhibit the same eccentricities. Making use of such discovered idiosyncracies leads to extra work in reprogramming when transferring the work to another computer or when an updated compiler replaces the old one.

1.5 PERMISSIBLE STRUCTURES

As part of the structure of a Basic JOVIAL *program*, nothing is permitted by unstated implication. If it is not prescribed by this document (or other documentation in the case of system-dependent features), it is not legitimate Basic JOVIAL code. In the matter of exceptions to prescribed forms, nothing is prohibited by innuendo. All exceptions are explicitly stated.

The document is to be taken as a unit. All sections, all figures, and the index-glossary are interrelated. For instance, the full meaning of *integer: :variable* is obtained from a synthesis of Sections 3.2, 3.2.1, 3.2.2, and 3.2.4, with due consideration for the exception stated in Section 3.2.4.

CHAPTER 2. ELEMENTS

2.1 INTRODUCTION

A *program* written in JOVIAL consists, basically, of *statements* and *declarations*. The *statements* specify the computations to be performed with arbitrarily named data. There are both *simple:statements* and *complex:statements*, which can be grouped together into *compound:statements*. Among the *declarations* are *data: :declarations* and *processing:declarations*. The *data:declarations* name and describe the data on which the *program* is to operate, including inputs, intermediate results, and final results. The *processing:declarations* generally contain *statements* and other *declarations*. They specify computations, but they differ from *statements* in that the computations must be performed only when the particular *processing:declaration* is specifically invoked by *name*. In addition to *statements* and *declarations* there are *directives* by means of which the compiler is caused to change its interpretation of certain structures in the *program*. The *statements*, *declarations*, and *directives* are composed of *symbols*, which are the words of the JOVIAL language. These *symbols* are in turn composed of the *signs* that constitute the JOVIAL alphabet.

The general order in which the elements of a *program* are introduced in the preceding paragraph represents the general order in which one looks up definitions when trying to clear up a question. The definitions in this manual are introduced, however, in the opposite order. Such arrangements lead to complaints that one must "read the book backwards." This comment arises from the process of looking up a form in the table of contents, turning then to the late chapter where it is defined in terms of earlier defined forms. These, more elementary, forms are then found, via the table of contents, in an earlier chapter. And so forth. Nevertheless, the document is arranged for the use of a reader rather than for reference. Difficult as this may make referencing, the opposite arrangement is much more difficult for a reader.

An index is included which, hopefully, facilitates reference. The index should answer many questions directly. It carries one quickly back through the chain of definitions until the question is answered or until the reader needs more details, to which he is directed through the section numbers. The index also contains references to structures that are not discussed in this manual. All such references are preceded by a daggar (†). These other structures are part of the J3 language, but not of Basic JOVIAL. The section numbers in such cases refer to TM-555/002 rather than to this document.

2.2 SPACES AND *SPACES*

It is important to distinguish between a *space*, an element of JOVIAL, and a space, an element of our descriptive language. JOVIAL is written using *symbols*, the words of the language. The *symbols* are composed of *signs*, the

elements of the JOVIAL alphabet. In general, *symbols* do not contain *spaces*. The exceptions are pointed out in Sections 2.5 (*comment*) and 2.6.3 (*hollerith: :constant*). In general, *symbols* are separated by *spaces*. Again the exceptions are noted (Section 2.7), but these exceptions are permissive--it is always correct to put *spaces* between *symbols*, except that it is never permitted to put a *space* after the + or - denoted by the word *signed* (see Section 2.6.1).

In defining and explaining *signs* and *symbols*, any spaces included in the metalanguage formulas are not meant to be included in the definition. The phrase "string of" implies that there are to be no *spaces* between the elements strung together. Similarly, phrases such as "followed by," "enclosed in," and "separated by," imply that there are to be no *spaces* between the elements concerned. This is the situation (except where explicitly stated to be different) up to Section 2.7. In Chapter 3 and beyond, the opposite view is maintained with respect to these phrases. In Sections 2.7 and 2.8 the transition is noted and forms are explained that don't quite fit either the new rule or the old one.

2.3 *SIGNS*, ELEMENTS OF THE JOVIAL ALPHABET

Sign means a *letter*, a *numeral* or a *mark*.

Letter means one of the 26 letters of the English alphabet, written in the form of a roman capital.

Numeral means one of the ten arabic numerals Ø, 1, 2, 3, 4, 5, 6, 7, 8, or 9.

Octal:numeral means one of the following eight *numerals*: Ø, 1, 2, 3, 4, 5, 6, 7.

Mark means one of the 12 marks, each associated with a name or names in parentheses, in the following list:

1.	+	(*plus:sign*)
2.	-	(*minus:sign*)
3.	*	(*star*)
4.	/	(*slash*)
5.		(*space, blank*)
6.	.	(*period, decimal:point*)
7.	,	(*comma*)
8.	=	(*equals:sign*)
9.	(	(*left:parenthesis*)
10.	)	(*right:parenthesis*)
11.	'	(*prime*)
12.	$	(*dollar:sign*)

2.4 *SYMBOLS*, THE WORDS OF JOVIAL

The *symbols* or words of the JOVIAL language are composed of strings of *signs*, in some cases a single *sign*. Most *symbols* do not contain *spaces*. In fact, *spaces* serve to separate *symbols* from one another. In the definitions of *symbols*, the phrase "enclosed in *parentheses*" means having a *left:parenthesis* on the left and a *right:parenthesis* on the right without any intervening *spaces*. If the input medium is cards (or card-images or other "unit records"), each symbol must be completely contained on one card.

Symbol means one of the following expressions:

1. *primitive*
2. *constant*
3. *loop:variable*
4. *abbreviation*
5. *name*
6. *ideogram*
7. *comment*

The above definition contains a categorical listing of all JOVIAL *symbols*, but *primitive* and *ideogram* have reference to the way these *symbols* are written rather than their use in constructing *programs*. These two categories can be regrouped in ways that are more suggestive of their roles in the language.

Those *symbols* that are *primitives* or *ideograms* include the categories in the following list, which is not exhaustive:

8. *arithmetic:operator*
9. *relational:operator*
10. *logical:operator*
11. *functional:modifier*
12. *bracket*

2.5 *PRIMITIVE, NAME, LOOP:VARIABLE, ABBREVIATION, IDEOGRAM, COMMENT*

The following list exhibits all the *primitives* of the Basic JOVIAL language:

ABS	DIRECT	IF	NQ	STOP
AND	END	ITEM	NWDSEN	SWITCH
ASSIGN	ENT	JOVIAL	OR	TABLE
BEGIN	EQ	'LOC	OVERLAY	TERM
BIT	FOR	LQ	PROC	TEST
BYTE	GOTO	LS	'PROGRAM	
CLOSE	GQ	NENT	RETURN	
DEFINE	GR	NOT	START	

A *primitive* is a *symbol* consisting, usually, of two or more *letters* and having a specific meaning in the JOVIAL language. In the above list there are two *primitives* that begin with the *prime*. This is in accordance with a policy of requiring the spelling of any new *primitive* added to the language to begin with this *mark*. The purpose is to avoid outlawing any previously written *programs* by preventing the possibility of the new *primitive* being identical to any *name*.

A *name* is a string of two through six *letters* and *numerals* with the following characteristics:

1. It has no more than ψ *signs*.
2. It is not identical to any *primitive*.
3. It begins with (the leftmost *sign* is) a *letter*.
4. It is not identical to any of the words in the following list:

ALL	FILE	MODE	OUTPUT
ARRAY	IFEITH	ODD	POS
CHAR	INPUT	OPEN	SHUT
ENTRY	MANT	ORIF	STRING

The above 16 words are *primitives* in the full JOVIAL language, but not in this subset. Nevertheless, even in Basic JOVIAL, the above words must not be used as *names*.

Loop:variable. Any single *letter* can be used as a *loop:variable*. It is the context in which it is used that characterizes it as a *loop:variable*. A *loop:variable* is often called by other terms such as for-variable or single-letter subscript.

Abbreviation. Several *letters* are used, standing alone, as *abbreviations*. The meaning of an *abbreviation* depends on context. Those *letters* that may be used as *abbreviations* are not exhibited here, but are shown and explained in connection with the forms in which they can occur.

Ideogram means a string of *marks* having meaning in JOVIAL. Each of the twelve *marks* except the *space* and the *prime* is also an *ideogram*. Following are listed the 18 Basic JOVIAL *ideograms*:

+	=	(/
-	(	/)
*	)	(*
/	$	*)
.	**	($
,	''	$)

Comment means two *primes* followed by a string of *signs* followed by two *primes*. The string of *signs* between the two sets of doubled *primes* may contain *spaces*. It must not contain two *primes* in succession; the last *sign* before the second set of two *primes* must not be a *prime*; and the string of *signs* must not contain $ except in the following two *ideograms*:

5. ($
6. $)

2.6 *CONSTANT*

Before proceeding with the definition of *constant* it is necessary to define certain adjectives and adverbs that are used to denote attributes of *constants*, *variables*, *functions*, and certain other expressions.

2.6.1 ADJECTIVES APPLYING TO JOVIAL

Hollerith means having values that are strings of *signs*, each *sign*, if within a computer, being represented in a manner dependent on the particular computer. Additional system-dependent characters may be included in the *hollerith* set. These additional characters may occur within *direct:code* exclusive of *direct:assigns*, or they may occur within *hollerith:constants*.

Integer, as a noun, means a numeric value that is represented as a whole number without a fractional part, but that is treated as if it had a fractional part with value zero to infinite precision. In this manual, precision means the number of bits to the right of the point in binary representations of numeric values.

Integer, as an adjective, means having the value of an *integer*.

Signed means being preceded by + or - without intervening *spaces*.

Fixed means having numeric values, within the computer, with a specific (system-dependent) degree of precision. *Fixed* applies to *formulas*, but not to *variables*, *constants*, or *functions*.

Floating means having numeric values represented within the computer by two numbers. These two numbers are the signicand, which carried the significant bits of the value, and the exrad, or exponent of the radix, which tells where the binary point is among the bits of the signicand or how far to right or left. In this manual, significant bits means the bits in a computer representation of a number without consideration of the reliability of any of the bits.

Octal means having values represented by *octal:numerals* and certain other *signs*. The value may be considered as an integer or as a bit pattern depending on context. *Octal* applies only to JOVIAL structures that are in the nature of *constants*.

Boolean means having one of two possible values, "true" and "false." *Boolean* applies to *formulas*, but not to *variables*, *constants*, or *functions*.

Status means having values that are, in essence, mnemonic labels. The representation, within a computer, of these values depends on context and not on the particular computer involved.

Literal means *hollerith* or *octal*.

Numeric means *integer* or *fixed* or *floating* or *octal*.

Having defined the above adjectives, it is now possible to use and understand certain terms without explicit definition. For instance, if *hollerith:constant*, *floating:constant*, etc., are defined, the meaning of *constant* is clear. Similarly, if *variable* is defined, the meanings of *status:variable*, *integer: :variable*, etc., are clear.

2.6.2 *OPTIONAL, OPTIONALLY, NUMBER, SCALE*

Optional means, with respect to the noun element to which it is applied, that the element may be present or absent. For example, *optional:signed:numeral* followed by *letter* means one of the following three forms:

1. + *numeral letter*
2. - *numeral letter*
3. *letter*

Optionally means, with respect to the adjective to which it applies, that the adjective may apply or not. For example, *optionally:signed:numeral* followed by *letter* means one of the following three forms:

4. + *numeral letter*
5. - *numeral letter*
6. *numeral letter*

Number means a string of *numerals*. If a *number* stands alone as a *symbol*, it has the conventional integral constant value.

Scale means a *number* in certain positions as indicated below.

2.6.3 THE STRUCTURE OF *CONSTANTS*

Integer:constant means a *number*. An *integer:constant* is a JOVIAL *symbol*. It has a numeric value given by reading it as a mathematical symbol. Its value must be representable in no more than π-1 bits.

Floating:constant means one of the six structures in the following list (as explained in Section 2.2, *spaces* are not permitted):

1. *number* .
2. *number* . *number*
3. . *number*
4. *number* . E *optionally:signed:scale*
5. *number* . *number* E *optionally:signed:scale*
6. . *number* E *optionally:signed:scale*

Examples of *floating:constants*:

7. 3.14159
8. 56789.E-3

Octal:constant means the *letter* O followed by a *left:parenthesis* followed by a string of *octal:numerals* followed by a *right:parenthesis*. Examples of *octal:constants*:

9. 0(2Ø2Ø2)
10. 0(12345670)

The value of an *octal:constant* is *literal* or *numeric* depending on context. If *literal*, the value is the pattern of bits represented, three bits per *numeral*, by the string of *octal:numerals*. If *numeric*, the value is the integer represented, in octal, by the string of *octal:numerals*.

Hollerith:constant means a *number* followed by the *letter* H followed by a *left:parenthesis* followed by a string of *signs* and system-dependent characters followed by a *right:parenthesis*. The value of the *number* must be no more than 8 and must correspond to the number of characters between the *parentheses*. The value of a *hollerith:constant* is the string of characters, represented within the computer in *hollerith*. The string of characters between the *parentheses* may include *spaces*. Examples:

11. 6H(TR CD)
12. 5H(HLRTH)

Status:constant means either a *letter* or a *name* enclosed in *parentheses* and preceded by the *letter* V. Following are two examples of *status:constants*:

13. V(A)
14. V(POOR)

The value of a *status:constant* depends entirely on context. In each context the *status:constant* is associated with a *status:item:name*. The *status:constants* associated with each *status:item:name* must differ among themselves, but they need not be different from those associated with other *status:item:names*. Indeed, the value of a *status:constant* associated with one *item:name* may be different from the value of that same *status:constant* when associated with a different *item:name*. Aside from the rules stated in this paragraph, the uniqueness of *names* and *loop:variables* required elsewhere does not apply to the interiors of *status:constants*.

2.6.4 COMPUTER REPRESENTATION OF *CONSTANTS* AND *VARIABLES*

The representation specified in this section is concerned only with the external effect. That is, internal representation may be different so long as the results produced by the *program* and the computer impinge on the environment precisely as they would if the representation were the same as specified here. Thus, it is likely that no special precautions are needed in assigning the value of one *variable* to another, similarly defined, but that the compiler must provide extra instructions to insure compatibility with the specified representation when utilizing or setting a partial bit-string of an *item*, as occurs when the programmer uses the *symbols* BIT and OVERLAY.

Integer nonpacked *items* and *constants* are represented by π-1 bits representing the magnitude of the value (Ø for Ø, 1 for 1, regardless of sign), preceded by one bit representing the sign (Ø for positive, 1 for negative). This representation is independent of the size of an *item* or *constant*, except that size must not exceed the limits set by the representation. The size of the magnitude of an *integer:constant* is π-1 less the number of zeros between the sign bit and the leftmost significant bit.

Example:

1. 1ØØ1100 (π-5 zeros)

The above example represents an *integer:constant* of value -12 having a magnitude size of 4. The bits immediately following the sign bit--numbering π-1 minus the size of the magnitude--are known as filler bits.

An *integer* packed *item* is represented by 1 sign bit (if the *item* is defined as signed) followed by an appropriate number (in accordance with the size, but not exceeding π-1) of magnitude bits. The meanings of Ø and 1 are as specified for non-packed *items*. *Floating:constants* and *floating:items* are represented by two numbers, the signicand (significant digits) and the exrad (exponent of the radix). The signicand consists of λ-1 bits representing the magnitude, and one bit representing the sign. The exrad consists of υ-1 bits

representing the magnitude, and one bit representing the sign. The meanings of Ø and 1 are as specified for non-packed *integer:items*.

No variation is permitted in the sizes associated with *floating:items*. If a *floating:item* in a *table* does not fill the *words* it occupies, an implementer may permit its position to vary in packed *tables* or *defined:entry:tables*.

In certain contexts, such as *literal:assignment:statements* and *literal:item::declarations*, *octal:constants* are considered to be *literal:constants*. In such a context an *octal:constant* is limited in size by the associated *literal::item*. The *octal:constant* may be large enough that the bit pattern it represents is as long as the number of bits in the representation of the *literal:item*. For instance, if *literal:items* are represented by 7 bits per *sign* and the context is a *statement* assigning a value to an 8-byte *literal:item* the *octal:constant* may contain as many as 19 *octal:numerals*, but its value must not be greater than 3 followed by 18 7's. If, in this context, the *constant* contains fewer than 19 *octal:numerals*, its value is prefaced with enough zeros to make a 56-bit pattern. The value is then represented as the pattern of bits corresponding to the string of *octal:numerals* in accordance with Table III.

Table III. Bit Patterns for *Octal:numerals*

Octal:numeral	Ø	1	2	3	4	5	6	7
Bit Pattern	ØØØ	ØØ1	Ø1Ø	Ø11	1ØØ	1Ø1	11	111

In other contexts, an *octal:constant* may contain no more than π *octal:numerals* and is represented as a pattern of π bits (leading zeros supplied if needed) in accordance with Table III.

Hollerith values are represented by bit strings, ω system-dependent bits per character or byte. *Hollerith* values must not be greater than 8 bytes in length.

Status:constants are represented as unsigned integers containing just enough bits to specify the value. Values are assigned consecutively starting with zero for the first *status:constant* in each *status:item:description*. A *status::item* is represented by a bit string just long enough to contain the largest associated *status:constant* value, if there is no *number* to declare the length of the bit string. If there is such a *number*, it must be no greater than π-1 and the bit string is as long as specified.

2.7 TRANSITION

All the *symbols* of Basic JOVIAL have now been explained, at least so far as their structure is concerned. Some meanings have also been explained, but others are made clear only as the use of the *symbol* in larger constructions is discussed.

In Chapter 3 and those that follow, such phrases as "string of," "followed by," "enclosed in," and "separated by" imply that *spaces* are permitted and may be required between the elements concerned. In writing a *program* all the *symbols* are to be separated by one or more *spaces* except that, if the meaning is still clearly the same, a *space* may be omitted. This means that, in general, *spaces* are required between *primitives*, *names*, *loop:variables*, *abbreviations*, and *constants*; but not required between an *ideogram* and another *symbol*. Note that . is an *ideogram* when used as a *period* following a *name* in certain situations (see Sections 3.4 and 3.5.5, for example), but not when used as a *decimal:point* in writing *constants* (Section 2.6.3). Similar remarks concerning + and - might be made, but no ambiguity results from disregarding such commentary.

Examples:

```
1.      BEGIN  GOTO
2.        3E2  7E5
3.         IF  'LOC
4.         P=Q+5$
```

There are exceptions to the general rule: (1) *spaces* may be omitted between a *primitive* or *abbreviation* and a following *constant* that begins with a *decimal:point*; (2) *spaces* may be omitted between a *constant* that ends in a *decimal:point* and a following *primitive* that does not begin with E or A.

Examples:

```
5.      BEGIN.5    .6    1.3    2.    END
6.      IF ALPHA     EQ    7.OR.3Ø2     LQ BETA$
```

In the metalanguage formulas to follow, the metalanguage symbol θ appears between *symbols* whenever a *space* is permitted or required in a corresponding JOVIAL structure. Whether or not the *space* is required depends on the particular *symbols* to be separated, in accordance with the above rules. In examples, *spaces* are not necessarily shown if not required.

A *comment* may replace any one or more of the string of *spaces* between *symbols* without altering the meaning of the structure except in the case of a *define::directive*, which is explained in the next section. A *comment* must not be used to replace a *space* within a *symbol* such as a *literal:constant* or another *comment*.

(page 24 blank)

A *comment* is only for the edification of a programmer reading a listing of the *program*. It has no effect upon the outcome of compilation.

2.8 *DEFINE:DIRECTIVE*

This structure is explained at this point because it fits neither rule concerning the use of *spaces* and *comments*.

Define:directive means a structure of the following form:

1. **DEFINE** θ *name* θ '' *constant* '' θ **$**

There must not be any *comments* between the *name* and the second '' *symbol*. *Comments* are permitted, however, at the other two points indicated by θ. The purpose of the *define:directive* is to permit a *name* to be used instead of the quoted *constant* at subsequent *points* in the *program*. Wherever such a "defined" *name* is used it is effectively replaced by the quoted *constant* with the following exceptions:

2. as part of a *status:constant*
3. as part of a *literal:constant*
4. as part of a *comment*

A *name* may be redefined by the use of another *define:directive* for the same *name* at subsequent point in the *program*, but it cannot be "undefined." That is, once a *name* has been given a definition for a particular *program* there is no device or language structure whereby it may be returned to the pool of unused *names* or to the usage it had before its first *define:directive*.

Note that *primitives* must not be redefined by the use of *define:directives*.

Examples of *define:directives:*

```
5.  DEFINE GOOD    ''V(GOOD)''$
6.  DEFINE PI      ''3.14159''$
7.  DEFINE AA      ''3H(ABC)''$
```

BLANK PAGE

CHAPTER 3. *STATEMENTS*

3.1 INTRODUCTION

A JOVIAL *program* consists of a string of *statements* and *declarations* that specify rules for performing computations with sets of data.

The basic elements of data, called *items*, are named to distinguish one from another. Sometimes a *name* applies to a group of *items*, requiring indexing to tell one member of the group from another. Several named groups may be subsumed under another group, which is known as a *table* and which is itself named.

The values of *items* and other data can be changed in various ways. A data element whose value can be changed by means of an *assignment:statement* is known as a *variable*. Among the JOVIAL *primitives* are some, known as *functional: :modifiers*, that can be applied to an *item:name*, thereby designating only a part of the *item* to be considered, for the moment, as a *variable*. Another *functional:modifier* can be used to group the *items* of a *table* together, the group being then considered a single *variable*.

The value to be given a *variable* is specified in an *assignment:statement* by means of a *formula*, which can be a *constant*, a *variable*, or a *function*. In case of *numeric:formulas*, arithmetic combinations of *formulas* are also *formulas*.

3.2 *VARIABLES*

Variables can be named and described in *item:declarations*, which declare and describe *items* of one kind or another. *Declarations* are discussed in Chapter 4. They can describe these *named:variables* in terms of the adjectives defined previously and listed below:

1. *hollerith*
2. *integer*
3. *floating*
4. *status*

The collective adjectives previously defined also apply. A *literal:variable* means a *hollerith:variable*. A *numeric:variable* means an *integer:variable* or a *floating:variable*. *Named:variables* can also be subdivided into *simple: :variables* and *indexed:variables*.

3.2.1 *SIMPLE:VARIABLES*

Simple:variable means the *name* of an *item* declared by an *item:declaration* not associated with any *table:declaration*. The adjectives that characterize the *variable* depend on the type description in the *declaration*. Example of a *simple:variable:*

1. ALPHA

3.2.2 *INDEXED:VARIABLES*

The notion of recursive definitions must now be introduced. *Indexed:variables* are defined in terms including the word *index*. *Index* is defined in terms of *formulas*, which are defined later in terms of *variables*, including *indexed:variables*. This is not to be interpreted as a circular definition with enigmatic meanings, but rather as a simple way of indicating how structures of any required complexity may be built up.

An *index* means a *numeric:formula* or a string of two *numeric:formulas* separated by a *comma*. Each *formula* in the string is known as a component. Each time an *index* is evaluated, each component must yield a positive value or zero. If the value is not an *integer*, it is converted and truncated to an *integer*. If the *formula* comprising a component of an *index* can be written in the following form:

1. *numeric:formula* θ + θ *constant*

the *numeric:formula* part must not be negative. Each component must also, of course, be small enough to specify an actual element of the data structure to which it applies.

Indexed:variable means a structure of the following form:

2. *name* θ ($ θ *index* θ $)

where *name* is the *name* of an appropriately declared *item* and the *index* consists of a single component. The *index* in the above structure serves to specify a particular value from a *table* of values. Example of an *indexed:variable*:

3. BETA($GAMMA$)

3.2.3 ADDITIONAL *VARIABLES*

As descriptions of *variables*, the terms *floating* and *status* may only describe *named:variables*, that is, *simple:* or *indexed:variables*. *Hollerith* and *integer*, however, may be applied to other *variables*, as explained in the following sections.

3.2.4 *INTEGER:VARIABLES*

Following is a list of the structures that, along with *named:integer:variables*, are also *integer:variables*. None of the following may ever be negative:

1. *loop:variable*
2. BIT θ ($ θ *index* θ $) θ (θ *named:variable* θ)
3. NENT θ (θ *name* θ)

BIT must not be applied to *floating:items*.

The two *primitives* in the above list are known as *functional:modifiers*. In the form with NENT, the *name* must be the *name* of a variable-length *table* or of an *item* belonging to a variable-length *table*. This *variable* designates the number of *entries* of the indicated *table*. Values less than zero or more than the declared maximum are undefined. The value before being set, as by an *assignment:statement*, depends on the system.

The form with the *functional:modifier*, BIT, designates the unsigned integer value represented by the string of bits, or a segment of the string, used in the machine encoding of the *simple:* or *indexed:variable*, but excluding any sign bit with unsigned *items* and any filler bits with nonpacked or medium-packed *items*. The number of bits in a *named:variable* is determined from its *declaration*. These bits are numbered from the left starting with zero. The *index* used with the BIT *modifier* may have two components, in which case the first component designates the first bit of the segment and the second component designates the number of bits in the segment. It is required, of course, that these be compatible with the size of the *item*. The second component must not have the value Ø. If only one bit is wanted, a one-component *index* may be used, indicating which one.

For <u>signed</u> nonpacked *integer:items* of size n, the sign bit is bit Ø and bits 1 through n-1 are the rightmost n-1 of the π-1 magnitude bits. For example, in a signed *simple:integer:item* of size 5 with value -12, bits Ø through 4 are 111ØØ and there are π-5 zeros between bits Ø and 1.

For <u>unsigned</u> nonpacked *integer:items* of size n, bits Ø through n-1 are the rightmost n of the π-1 magnitude bits. For example, in an unsigned *simple:integer:item* of size 4 with value 13, bits Ø through 3 are 11Ø1 and there are π-4 fillers before bit Ø.

3.2.6 *LITERAL:VARIABLES*

The following structure, in addition to the *named:literal:variable*, is also a *literal:variable*:

1. BYTE θ ($ θ *index* θ $) θ (θ *named:literal:variable* θ)

The BYTE *modifier* functions in a manner entirely analogous to the operation of the BIT *modifier*. The machine language representation of a *named:literal :variable* is a string of bytes--each byte itself a string of ω bits representing a single character. The bytes of an n-byte *hollerith:item* are indexed from left to right from Ø through n-1. The one- or two-component *index* used with the BYTE *modifier* indicates a substring of the bytes representing the value of the *item* modified. The first component of the *index* indicates the initial byte of the substring. For a two-component *index*, the second component indicates the number of bytes in the substring. For a one-component *index*, the length of the substring is implicitly one byte. The BYTE *variable* is defined only if the *index* on the BYTE *modifier* indicates a substring of bytes within the byte range of the *item*. A byte-string of zero length is undefined.

Example:

2. BYTE ($ I, 2 $) (MESSAGE ($ K $))

3.2.8 *ENTRY:VARIABLES*

A *table* (discussed at greater length in Chapter 4) is an ordered set of *entries*, indexed from Ø through n-1 for an n-*entry table*. Each *entry* is a set of related *items*; related, perhaps, only by having been declared to constitute a single *table*. An *entry:variable* is an agglomeration of the values of the *items* constituting an *entry* of a *table*. Its value depends on both the structure of the *entry* and the values of the *items* forming the *entry*. This value may be denoted by Ø if all the bits in the *entry* have the value Ø. Otherwise there is no JOVIAL *constant* that can denote the value. The form of the *entry:variable* is:

1. ENT θ (θ *name* θ ($ θ *index* θ $) θ)

in which the *name* is the *name* of a *table* or of one of the *items* of the *table* and the *index* consists of just one component, designating which *entry*.

3.3 *FORMULAS*

Formulas are the means for expressing values. Hence *variables* and *constants* are also *formulas*. The adjectives that characterize *variables* may also be applied to *formulas*. An important kind of *formula* is the *function* or *function: :call*.

3.3.1 *FUNCTIONS*

A *function* means one of the following structures:

1. *function:name* θ (θ *actual:input:parameter:list* θ)
2. *function:name* θ (θ)

A *function* is also known as a *function:call*. *Actual:input:parameter:list* is explained in Section 3.5.5 in connection with *procedure:call:statements*. Even if the *actual:input:parameter:list* is missing the *parentheses* are needed to identify the *name* as a *function:name*. The *name* refers to a *function: :declaration*, described in Section 5.6. The *actual:parameters* must conform to the *formal:parameters* of the corresponding *function:declaration* in the same manner as explained for *procedure:call:statements*.

A *function* has a value that is *hollerith*, *floating*, etc. and that derives from the computations specified in the *function:declaration* that defines it.

Examples:

3. RANDOM ( )
4. COS (ALPHA)
5. GRADE (FINAL , MID + (T1 + T2 + T3) / 2)

Any *function* that directly or indirectly changes the value of a *variable* must not occur in the same *formula*, *clause*, or *simple:statement* (other than a *procedure:call:statement*) with that *variable*. If such a *function* occurs in a *procedure:call:statement* with the affected *variable*, they still must not occur in the same *parameter*, i.e., the same *formula*.

3.3.2 *LITERAL:FORMULAS, STATUS:FORMULAS, ENTRY:FORMULAS*

A *hollerith:formula* means one of the structures in the following list:

1. *hollerith:constant*
2. *hollerith:variable*
3. *hollerith:function*

A *literal:formula* means one of the following expressions:

4. *octal:constant*
5. *hollerith:formula*

A *status:formula* means one of the expressions in the following list:

6. *status:constant*
7. *status:variable*
8. *status:function*

An *entry:formula* means one of the following two structures:

9. Ø
10. *entry:variable*

The value of an *entry:variable* in which all the bits are zeros may be denoted by Ø in JOVIAL. No other value of an *entry:variable* may be denoted in any way in JOVIAL, hence the limited definition of *entry:formula*.

3.3.3 *NUMERIC:FORMULAS*

An *arithmetic:operator* means one of the *ideograms* in the following list, in which the meaning is given on the same line with each:

1.	+	addition
2.	-	subtraction
3	*	multiplication
4.	/	division
5.	**	exponentiation

A *numeric:formula* means one of the following structures:

6. *numeric:constant*
7. *numeric:variable*
8. *numeric:function*
9. NWDSEN θ (θ *name* θ)
10. NENT θ (θ *name* θ)
11. 'LOC θ (θ *name* θ)
12. 'LOC θ (θ *name* θ . θ)
13. + θ *numeric:formula*
14. - θ *numeric:formula*
15. (θ *numeric:formula* θ)
16. (/ θ *numeric:formula* θ /)
17. ABS θ (θ *numeric:formula* θ)
18. *numeric:formula* θ *arithmetic:operator* θ *numeric:formula*
19. *numeric:formula* θ (* θ *numeric:formula* θ *)

A *numeric:formula* containing one or more *arithmetic:operators* specifies the value arising from the computations described by the *formula*, in the familiar sense as defined by the notation of ordinary algebra, with a few exceptions as noted herewith. The forms with (/ and /) and with ABS denote the absolute value of the enclosed *formula*. Exponentiation is denoted by ** or by the (* and *) *brackets*, which, in the form given at the end of the above list, indicate the first *formula* raised to the power of the second, enclosed, *formula* subject to the precedence rules in the next paragraph. The *formula* indicating the power of the exponentiation must be in *integer* form; that is, it must contain no *floating:items*, *floating:constants*, or divisions.

Multiplication, denoted by *, and exponentiation must be explicitly shown. The unary operator + may be used although it is redundant. The unary operator - means negation. *Parentheses* and the other *brackets* perform their usual grouping function. At any level the order of operations is: negations, exponentiations, multiplications and divisions, and finally additions and subtractions. Within these categories operations are performed from left to right. Division by zero is undefined.

Note that since negation has high priority and subtraction low priority, the *formulas* listed at the left below have the values listed at the right:

```
20.     7 - 2 ** 2          3
21.     -3 ** 2 - 5         4
```

In *numeric:formulas*, each arithmetic operation involving a *floating:constant*, a *floating:variable*, or an intermediate result in *floating* form is performed in *floating* form. For instance, the sum of five *items*, the third one alone being a *floating:item*, involves one *integer* addition followed by three *floating* additions. An *index* component consisting of a *variable* plus or minus a *constant* is an exception; the addition or subtraction is done in *floating* form only if both the *constant* and *variable* are already in *floating* form. All calculations in *floating* form are truncated to λ-1 significant bits. Division of one *integer* by another *integer* yields a *fixed* intermediate result. If this *fixed* result is to be assigned to a *floating:variable* or used in further computation, the fractional bits are carried along. If it is to be assigned to an *integer:variable*, the fractional bits are truncated. The number of fractional bits depends on the sizes of the *integers* in the computation (see Section 3.3.4). Necessary conversions between *floating* and *fixed* or *integer* are carried out automatically.

Exponentiation is carried out in *floating* form if the base is *floating*. Otherwise, the scaling rules (Section 3.3.4) for multiplication and division apply--as many as π-2 multiplications followed by, at most, one division.

In the forms with NWDSEN and NENT the *name* must be a *table:name* or the *name* of an *item* belonging to a *table*. NWDSEN indicates the number of words per *entry* of the referenced *table*, a constant integer value. NENT indicates the number of *entries* of the referenced *table*, another integer. As mentioned previously, the application of NENT to a *variable:length:table* yields an *integer:variable*. If the *table* is of rigid length, the use of NENT yields a *formula*, a constant integer value, but not a *variable*.

In the forms using 'LOC, the *name* must be a *program:name*, a *statement:name*, a *table:name*, or an *item:name*. The *statement:name* or *program:name* must be followed by a *period*; the *table:name* or *item:name* must not. The value of the 'LOC *formula* is a nonnegative integer equal to the number of the first (or only) basic addressable unit that is allocated for the *simple:item*, the *table*, the *named:statement*, the *program*, or the first occurrence of the *table::item*. The specified basic addressable unit is not one allocated to any

associated control register that may precede the *item*, *table*, or *program*.

Examples of *numeric:formulas*:

```
22.     ALPHA     +      BETA
23.     GAMMA     /     (DELTA  ($ I $) ** (/XX - YY/)    )
24.     EPSILON         (* SIN   (PHI ** 2)   **2 - COS (PHI ** 2)   ** 2 *)
25.     ('LOC(ZETA)        + NENT(TBL) * NWDSEN(TBL))/2
```

3.3.4 SCALING OF *FIXED* INTERMEDIATE RESULTS

When a computation is not to be done in *floating* form, the following rules apply to the scaling of the results of one operation using two numbers. Following application of these rules to a complete *formula* the required assignment or comparison is done. In assigning and utilizing nonpacked *items*, all the bits in the representation are carried along. This permits greater flexibility to the sophisticated programmer, but opens the way to divide faults and overflows. The action in case of divide fault or overflow is system dependent:

1. Representation and Truncation. Intermediate results are represented by π bits: the sign, filler bits, integer bits (the number determined as described below), and fraction bits (the number determined as below). When the number of bits (determined as below) exceeds π-1, excessive bits are truncated. Truncation is performed first on least significant bits of the fraction part, then, if necessary, on most significant bits of the integer part.

2. Terminology. By the time *data:declarations* are translated to the binary level, every *integer* value is stated (or understood) to be signed or unsigned. The size of each *item* is given as the number of bits allotted to that *item*. This must be a positive number and includes the sign for signed *items*. However, in these rules for scaling, size is considered exclusive of sign. For unpacked *items* the size is determined from the *declaration* or other definition, not from the representation, which always has π-1 magnitude bits. Similarly, the size of a *constant* is measured from the first 1 bit on the left. This discussion is concerned with two types of operands, I-type and A-type. *Integer:variables*, *integer:constants*, and many intermediate results derived from them are known as I-type. *Fixed* results of division of *integers* and results of computation with these *fixed* results are called A-type. Various attributes of the operands entering a computation and of the result are designated here by two-character codes. The meanings of the first characters of these codes are as follows:

S	size, exclusive of sign
I	bits in the integer part
A	bits in the fraction part

The above characters are also used alone with these meanings when the discussion is not concerned with which element exhibits the characteristic. The meanings of the second characters are as follows:

1	first operand
2	second operand
N	numerator
D	denominator
I	the *integer* operand (implies there is only one)
A	the *fixed* operand (implies there is only one)
R	the result

The number of fraction bits, A, may be positive or zero. It is understood to be zero for *integers*; however, all (zero) bits to the right of the point are understood to be significant. The number of integer bits, I, is not stated but is equal to S-A. In the following rules IR and AR are determined independently. Then SR (equal to IR+AR) is used to determine the need for truncation.

3. Addition and Subtraction. If both operands are I-type, the result is I-type. Otherwise the result is A-type.

- IR = 1 + maximum (I1, I2)
- If A1 = A2, then AR = A1.
- If one operand is I-type, then AR = AA.
- In the remaining cases,

  AR = 1 + minimum (A1, A2).

4. Multiplication. If both operands are I-type, the result is I-type. Otherwise the result is A-type.

- IR = I1 + I2
- AR = A1 + A2

5. Division. The result is A-type, even if both operands are I-type.

- If both operands are I-type,

  IR = IN

  AR = π-1-IN

- If the denominator is A-type,

  IR = IN + AD.

- If the numerator is A-type,

  AR = ID + AN.

- If the numerator is I-type and the denominator A-type,

  AR = 2(ID) + AD - 1.

3.3.6 *RELATIONAL* OPERATIONS

A *relational:operator* is the means of expressing a relation between two *formulas*. The relation is in the form of a proposition that may be either true or false. Hence the proposition is a *boolean:formula*. A *relational:operator* means one of the *primitives* in the following list:

1.	EQ	is equal to
2.	GR	is greater than
3.	GQ	is greater than or equal to
4.	LQ	is less than or equal to
5.	LS	is less than
6.	NQ	is not equal to

In the above list the meaning of each *relational:operator* follows it on the same line. The effect of a *relational:operator* is to state that the *formula* on its left stands in the indicated relation to the *formula* on its right. The meaning of such a proposition is fairly obvious in the case of *numeric::formulas*, and its truth is determined by an arithmetic comparison. Between *status:formulas* the truth of the relation depends on the numeric encoding of the values as unsigned integers.

3.3.7 PRECISION OF ARITHMETIC COMPARISONS

A *fixed* or *integer* value is converted to *floating* form for comparison with a *floating* value. In comparing an *integer* with a *fixed* value, the comparison is carried out to AA bits (see the paragraph on terminology in Section 3.3.4) after the point. In comparing two A-type values, the comparison is carried out to the smaller of A1 and A2 bits after the point.

3.3.8 *BOOLEAN:FORMULAS*

A *boolean:formula* means one of the structures in the following list:

1. *numeric:formula* θ *relational:operator* θ *numeric:formula*
2. *literal:formula* θ EQ θ *literal:formula*
3. *literal:formula* θ NQ θ *literal:formula*
4. *status:variable* θ *relational:operator* θ *status:formula*
5. (θ *boolean:formula* θ)
6. NOT θ *boolean:formula*
7. *boolean:formula* θ AND θ *boolean:formula*
8. *boolean:formula* θ OR θ *boolean:formula*

In the forms above with *literal:formulas*, the two *formulas* in the comparison must each consist of the same number of bytes.

The three *primitives*, AND, OR, and NOT, are known as *boolean:* or *logical:* *:operators*. Their meanings are illustrated in Table IV. In the heading of the table, p and q stand for *boolean:formulas*. The body of the table shows the values of the compound *boolean:formulas*, NOT q, p AND q, and p OR q, corresponding to the possible combinations of values of p and q. Ø means false and 1 means true. In *boolean:formulas* containing *logical:operators*, *parentheses* may be used to indicate the scope of the *operators*, as recursively shown by the structures in the list of *boolean:formulas*. Where precedence is not shown by *parentheses*, NOT takes effect first, then AND, finally OR. Within these categories, the sequence of operations is from left to right.

Boolean:formulas are evaluated from left to right. *Arithmetic:operators* must be applied before the *numeric:formulas* in which they occur can be compared by means of the *relational:operators*, which, in turn, are applied to determine elementary truth values before the *logical:operators* may combine them to determine the overall truth value of the *boolean:formula*. Evaluation proceeds from left to right, but only so far as is necessary to determine the truth value of the *formula*. If, at some point, the truth of the entire *formula* is not ascertained, but it becomes unnecessary to evaluate a certain portion of the *formula*, that portion is not evaluated. (See Section 6.5.3).

Table IV. Effect of the *Logical:operators*

p	q	NOT q	p AND q	p OR q
Ø	Ø	1	Ø	Ø
Ø	1	Ø	Ø	1
1	Ø	1	Ø	1
1	1	Ø	1	1

Examples of *boolean:formulas*:

```
9.      1.5 LQ  XX
10.     ALPHA  EQ  BETA  AND  NOT  LIT  LS  0(Ø377)
11.     (A+B  LS  C-D  OR X+Y  GR  2)  AND (Z  EQ  Y  OR  A  LS  C)
```

Consider the order of evaluation of Example 11 under two different conditions, first:

Get the current value of A (call this a).
Get the current value of B and add it to a (call this b).
Get the current value of C (call this c).
Get the current value of D and subtract it from c (call the difference d).
Compare b with d (assume b is less than d).
Get the current value of Z (call this e).
Get the current value of Y and compare it with e (assume Y equals e).
The value of the *formula* is "true" (quit evaluating).

Now try it again:

Get the current value of A (call this a).
Get the current value of B and add it to a (call the sum b).
Get the current value of C (call this c).
Get the current value of D and subtract it from c (call the difference d).
Compare b with d (assume b is <u>not</u> less than d).
Get the current value of X (call this f).
Get the current value of Y and add it to f (call the sum g).
Compare g with 2 (assume g is not greater than 2).
The value of the *formula* is "false" (quit evaluating it).

3.4 CLASSES OF *STATEMENTS*

Statements are the operational units of JOVIAL. They describe self-contained rules of computation, specifying manipulations of data, or, conditionally or unconditionally, sequencing of the execution of *statements*, or both.

In following sections the various kinds of *statements* are explained. Here, they are all listed. *Statement* means any of the expressions in the following list:

1. *independent:statement*
2. *named:statement*
3. *simple:statement*
4. *compound:statement*
5. *complex:statement*

Independent:statement means a *simple:statement* or a *compound:statement*.

Named:statement means the following expression:

6. *name* θ . θ *statement*

Statement:name means the *name* in the above expression. From the definitions of *statement* and *named:statement* it can be seen that a *statement* may have more than one *name*.

Example:

7. CEASE. DESIST. HALT. WHOA. STOP $

In the above example, from the space before any of the four *names* up to and including the *dollar:sign*, we have a *stop:statement*. Each of the four *names* is a *name* of this *statement*.

In the definitions of the various kinds of *statements* to follow, the *statements* are explained without *names*, but it is to be understood that they retain the defined characteristics when they are named. Thus a *stop:statement* remains a *stop:statement* whether or not it is also a *named:statement*. The following list exhibits three *named:statements*. The first line is also a *simple:statement*, the second line is also a *complex:statement*, and the third line is also a *compound:statement*:

8. S1 . STOP $
9. S2 . IF THETA EQ 45 $ XX = .7Ø7 $
10. S3 . BEGIN ALPHA = ALPHA + 1 $ BETA = GAMMA/ALPHA $ END

3.5 *SIMPLE:STATEMENTS*

Simple:statement means one of the expressions in the following list:

1. *assignment:statement*
2. *go:to:statement*
3. *test:statement*
4. *return:statement*
5. *stop:statement*
6. *procedure:call:statement*

3.5.1 *ASSIGNMENT:STATEMENTS*

Assignment:statement means one of the expressions in the following list:

1. *numeric:variable* θ = θ *numeric:formula* θ $
2. *literal:variable* θ = θ *literal:formula* θ $
3. *status:variable* θ = θ *status:formula* θ $
4. *entry:variable* θ = θ *entry:formula* θ $

Assignment:statements can be further characterized, in the obvious way, by means of the adjectives that occur in each of the above four expressions. For instance, *numeric:assignment:statement* means the expression on the first line of the above list.

An *assignment:statement* specifies that the *formula* to the right of the = *sign* be evaluated and that this value become the new value of the *variable* to the left of the = *sign*. It is permissible for the *variable* on the left to occur also in the *formula* on the right. In this case the old value of the *variable* is used in the calculations needed to evaluate the *formula*. A *function* may, of course, be included in the *formula*. However, the evaluation of a *function* must not involve side effects; that is, it must not change the values of other elements of the *assignment:statement* in which the *function* is embedded. In evaluation of *numeric:formulas*, rules have already been stated concerning conversions to compatible forms among *integer*, *fixed*, and *floating* values. Such conversions are also carried out where necessary in assigning the value as required in *numeric:assignment:statements*.

In executing *literal:assignment:statements* the value of the *formula* must be the same length as the *literal:variable* to which it is to be assigned.

If the *formula* on the right in a *status:assignment:statement* is a *status:constant*, it must be one of those appearing in the *declaration* that describes the *status:variable* appearing in the same *assignment:statement*. Otherwise there is no way for the compiler to associate a value with the *status:constant*.

In the *entry:assignment:statement* if the *entry:formula* on the right is an *entry:variable*, it must not differ in length (number of words) from the *entry: :variable* on the left.

3.5.3 *GO:TO:STATEMENTS*

A *sequence:designator* specifies a sequel in the sequence of *statement* executions. Normally the *statements* of a *processing:declaration* or of a *program* are executed in the order in which they are written. However, this normal execution order is modified by use of a *sequence:designator*, among other devices. A *sequence: :designator* means one of the two following expressions:

1. *name*
2. *name* θ ($ θ *index* θ $)

In the first of the above forms, the *name* must be the *name* of a *statement*, a *program*, a *close:declaration*, or a *switch:declaration*. In the second form the *name* may only be the *name* of a *switch:declaration*.

Go:to:statement means an expression of the following form:

3. GOTO θ *sequence:designator* θ $

A *go:to:statement* may interrupt the ordinary, listed sequence of *statement* executions, defining its successor explicitly by means of a *sequence:designator*. This interruption does not occur if the *sequence:designator* does not lead, directly or through a *switch*, to a *statement:name*, a *program:name*, or the *name* of a *close:declaration*, and the next *statement* executed is therefore the next listed. If the *sequence:designator* is, or leads to, the *name* of a *program* or of a *close:declaration*, the interruption may only be temporary, since a *program* or a *close:declaration*, upon execution, may be expected to return control to the next *statement* listed after the *go:to:statement* that invokes it. Finally, if the *sequence:designator* is, or leads eventually to, a *statement:name*, the interruption of the *statement* execution sequence is permanent, with the next *statement* executed being the one bearing the specified *statement:name*.

3.5.4 *TEST:STATEMENT, RETURN:STATEMENT, STOP:STATEMENT*

Test:statement means one of the expressions in the following list:

1. TEST θ $
2. TEST θ *loop:variable* θ $

Although a *test:statement* is a *simple:statement*, it may only appear within a *loop:statement* and its explanation depends on concepts pertinent to the *loop: :statement*. Its explanation is therefore postponed until the *loop:statement* is explained (Section 3.7.7).

The *return:statement* means RETURN followed by a *dollar:sign*. A *return:statement* indicates an operational end to a *close:declaration*, a *procedure:declaration*, or a *function:declaration*, and may thus appear only within one of these *processing:declarations*. It serves to terminate the execution of a *processing: :declaration* by transferring the *statement* execution sequence to the exit routine, which automatically follows the last listed *statement* of the *declaration*. A *return:statement* at the end of the *declaration* would be redundant. An exit routine, being an implied function, can have no *statement:name*, and, therefore, cannot be referenced in a *go:to:statement*.

The *stop:statement* means one of the expressions in the following list:

3. STOP θ $
4. STOP θ *statement:name* θ $

A *stop:statement* serves to halt the sequence of executions. It usually indicates an operational end to the *program* in which it appears. If the present *program* is being compiled as a closed subroutine (Section 6.1), the *stop:statement* without reference to a *statement:name* calls for the return of control to the external, calling *program*. If the environment includes an operating system and the present *program* is otherwise independent, such a *stop:statement* is compiled so as to return control to the operating system. If the computer halts without transferring control and if it is then restarted by some means, the execution sequence resumes with the next *statement* listed, or with the *statement* bearing the specified *statement:name* if one is given in the *stop:statement*.

3.5.5 *PROCEDURE:CALL:STATEMENTS*

An *actual:input:parameter:list* means one of the following two expressions:

1. *formula*
2. *actual:input:parameter:list* θ , θ *formula*

There is one minor exception needed to make this definition complete. A *status: :constant* is not permitted as one of the *parameters* in an *actual:input:parameter: :list*. The reason for this is that there is no place in the *list* or in the *statement* in which it occurs (a *procedure:call:statement*) for the *status: :variable* that would provide a value meaning for the *status:constant*.

An *actual:output:parameter:list* means one of the following two expressions:

3. *variable*
4. *actual:output:parameter:list* θ , θ *variable*

A *procedure:call:statement* means one of the expressions in the following list:

5. *procedure:name* θ $
6. *procedure:name* θ (θ) θ $
7. *procedure:name* θ (θ *actual:input:parameter:list* θ) θ $
8. *procedure:name* θ (θ = θ *actual:output:parameter:list* θ) θ $
9. *procedure:name* θ (θ *actual:input:parameter:list* θ = θ *actual:output:parameter:list* θ) θ $

A *procedure:call:statement* serves to call for the execution of a *procedure*, which is a self-contained process with a fixed and ordered set of parameters. A *procedure* is defined by a *procedure:declaration*. In general, a *procedure:call:statement* consists of a *procedure:name*, a set (possibly empty) of *actual:parameters*, and necessary delimiters. The *actual:parameters* of a *procedure:call:statement* must agree in type, number, and position with the *formal:parameters* of the *procedure:declaration* that bears the same *name*. In the *procedure:declaration* the *names* listed as *formal:parameters* are referenced elsewhere in the *declaration*.

Execution of the *procedure* is effected as if the values of those *formulas* that are *actual:input:parameters* are assigned to the *items* that are *formal:input:parameters* before execution and the values of the *formal:output:parameters* are assigned to the *variables* that are *actual:output:parameters* after execution. Consequently there must be compatibility between *formal:parameter:items* and the corresponding *actual:parameter:formulas* and *variables*, of the same nature as exhibited by *assignment:statements* (Section 3.5.1). *Actual:input:parameters* are evaluated and these values assigned to the corresponding *formal:input:parameters*, from left to right exactly as if directed by a sequence of *assignment:statements*. Following this the *procedure:body* is executed. Following this, if exit from the *procedure:body* is by a normal return, the values of the *formal:output:parameters* are assigned to the corresponding *actual:output:parameters*, from left to right just as if specified by a series of *assignment:statements*. If exit from the *procedure:body* is via an explicit jump to a global *name*, the *actual:output:parameters* are not assigned.

3.6 *COMPOUND:STATEMENT*

A *compound:statement* is a string of *statements* enclosed in the *brackets*, BEGIN and END. The enclosed *statements* may be named or not, *simple*, *compound*, or *complex*; and there may be *declarations* and *directives* included among them. In order to make the definition more precise, it is necessary to define a *statement:list*.

A *statement:list* is one of the expressions in the following list:

1. *statement*
2. *declaration* θ *statement:list*
3. *directive* θ *statement:list*
4. *statement:list* θ *declaration*
5. *statement:list* θ *directive*
6. *statement:list* θ *statement:list*

A *compound:statement* means the following expression:

7. BEGIN θ *statement:list* θ END

Example of a *compound:statement*:

8.
```
      BEGIN
      ALPHA = L $
SL1.  GOTO CL1 $
SL2.  INT(XØ , X1 = AREA) $
      END
```

3.7 *COMPLEX:STATEMENTS*

Complex:statement means one of the expressions in the following list:

1. *direct:statement*
2. *conditional:statement*
3. *loop:statement*

3.7.1 *DIRECT:STATEMENTS*

The *direct:statement* is a means for breaking out of the JOVIAL language within a *program* and writing some instructions in another language more directly related to the organization of the computer for which the *program* is being compiled. What is legal and meaningful within a *direct:statement* depends on the particular version of the compiler that is processing the *program*. For a precise definition of *direct:statement* it is necessary to make a few preliminary definitions.

Direct:assign provides access to the *variables* of a JOVIAL *program* from within a *direct:statement*. *Direct:assign* means one of the expressions in the following list:

1. ASSIGN θ A(Ø) θ = θ *named:variable* θ $
2. ASSIGN θ A() θ = θ *named:variable* θ $
3. ASSIGN θ *named:variable* θ = θ A(Ø) θ $
4. ASSIGN θ *named:variable* θ = θ A() θ $

In forms 1 and 2 above, the value of the *named:variable* is moved to the accumulator (the principal program-accessible register of the arithmetic unit). In forms 3 and 4 above, the value is moved from the accumulator to the *variable*. If A() is used, the accumulator contains a *floating* value. In other cases A(Ø) is used.

Direct:code means an essentially arbitrary string of JOVIAL *signs* not including the *symbol* JOVIAL, optionally interspersed with *direct:assigns*. More specifically, *direct:code* means one of the following expressions, but not including the *symbol* JOVIAL:

5. *signs*
6. *direct:assign*
7. *direct:code* *direct:code*

Direct:statement means:

8. DIRECT θ *direct:code* θ JOVIAL

Although *direct:code* is arbitrary so far as the definition of JOVIAL expressions is concerned, only certain configurations are meaningful. If the input medium is punched cards, specifications of meaningful *direct:code* probably involve positioning on the card. Because of this it is probably "safest" to prepare programs so that each *direct:assign* is on a separate card without other *direct::code* (except *spaces*) and so that there is no *direct:code*, besides *spaces*, on the cards containing the *symbols* DIRECT and JOVIAL.

3.7.2 *CONDITIONAL:STATEMENTS*

A *conditional:statement* means:

1. *if:clause* θ *independent:statement*

Remember that an *independent:statement* is a *simple:statement* or a *compound::statement*.

If:clause means one of the two following expressions:

2. IF θ *boolean:formula* θ $
3. *statement:name* θ . θ *if:clause*

The effect of a *conditional:statement* is that if the value of the *boolean::formula* of the *if:clause* is true, the *independent:statement* is executed; otherwise the *independent:statement* is skipped.

Presented below are two examples of *conditional:statements*:

```
4.  IF ALPHA - BETA LS 2 $ GOTO NEAR $
5.  IF BOOL EQ 1 $ LBL . BEGIN RANDOM (= BASIC) $ BASIC = BASIC
    ** 2 $ END
```

3.7.4 *LOOP:STATEMENTS*

The *loop:statement* provides for the iteration of an *independent:statement*. The iterations or repetitions of the *independent:statement* are controlled by means of one or more *loop:variables*, which are set up by *for:clauses*. The *loop:statement* consists of these *for:clauses* immediately followed by the iterated *independent:statement*. Remember that a *loop:variable* is a single *letter* in certain contexts. Those contexts are described below.

Complete:for:clause means the following expression:

1. FOR θ *loop:variable* θ = θ *numeric:formula* θ , θ *numeric:formula* θ , θ *numeric:formula* θ $ θ

The *complete:for:clause* defines a *loop:variable* to control the iteration of an *independent:statement* and that may be used as an *integer:variable* within the *statement*. The first of the three *numeric:formulas* is the initial value, given immediately to the *loop:variable* (in the sense of assignment to an *integer:variable*). The second *formula* provides an increment to be added to the *loop:variable* for each iteration. The third *formula* is a limit for iteration. After the *loop:variable* has been increased by the current value of the increment, it is compared with the current value of the limit. If it has not gone beyond the limit, execution of the *independent:statement* (the one controlled by the *for:clause*) is repeated. If the value of the *loop:variable* after incrementation is beyond the value of the limiting *formula*, the *independent:statement* is not repeated. "Beyond" means "greater than" or "less than," depending on the explicit sign of the increment. Specifically, if the second *formula* begins with a *minus:sign*, "beyond" means "less than;" otherwise "beyond" means "greater than."

Incomplete:for:clause means a *two:factor:for:clause* or a *one:factor:for:clause*. A *two:factor:for:clause* means the following expression:

2. FOR θ *loop:variable* θ = θ *numeric:formula* θ , θ *numeric:formula* θ $

The *two:factor:for:clause* defines a *loop:variable* with some measure of control over the iteration of an *independent:statement*. The first of the two *numeric:formulas* provides the initial value of the *loop:variable*. The second *formula* provides the increment to the *loop:variable* for each iteration of the *independent:statement*. There is no limiting value provided, and termination of the repeated executions must be provided by some other means.

One:factor:for:clause means the following expression:

3. FOR θ *loop:variable* θ = θ *numeric:formula* θ $

A *one:factor:for:clause* defines a *loop:variable* and gives it an initial value, but it does not cause any iteration.

Incomplete:for:clause:list means one of the following expressions:

4. *incomplete:for:clause*
5. *incomplete:for:clause* θ *incomplete:for:clause:list*

Incomplete:loop:statement means the following expression:

6. *incomplete:for:clause:list* θ *independent:statement*

Note that an *incomplete:loop:statement* is a *statement* and may therefore be preceded by a *statement:name* and a *period*. One example of an *incomplete:loop:statement* is the following:

```
7.      SL1.  FOR I = 1 , I $
              FOR J = I + 5 $
              BEGIN AA ($ J $) = BB ($ I $) $
                    J = 2 * I - 1 $
                    IF BB ($ I $) EQ Ø $ GOTO EXIT $
                    IF J GR 1ØØØ $ GOTO SL1 $
              END
```

Complete:for:clause:list means one of the following expressions:

8. *complete:for:clause*
9. *complete:for:clause* θ *incomplete:for:clause:list*
10. *one:factor:for:clause* θ *complete:for:clause:list*

Complete:loop:statement means the following expression:

11. *complete:for:clause:list* θ *independent:statement*

From the above we see that a *loop:statement* is a string of *for:clauses* followed by an *independent:statement*. Heading a *complete:loop:statement* there must be just one *complete:for:clause*, which may be preceded by *one:factor:for:clauses*.

3.7.5 USE OF *LOOP:STATEMENTS*

The effect of a *loop:statement* is to define a set of *loop:variables* and, usually, to execute an *independent:statement* repetitively. When all the *loop:variables* have been given their initial values, the *independent:statement* is executed. If all the active *loop:variables* have been defined by *one:factor:for:clauses*, there is no automatic repetition of the *independent:statement*. Otherwise, if execution of the *independent:statement* goes to completion without jumping out, it is iterated under control of the *loop:variables* as explained in Section 3.7.7.

Since a *loop:statement* is a *statement*, it may be part of the *statement:list* that forms part of a larger *loop:statement*. Such nesting of *loop:statements*,

in general, leads to repetition of the execution of the inner *loop:statement*, each execution of this inner *loop:statement* leading to repetitive executions of the *independent:statement* that forms its latter part.

Each *for:clause* defines or activates the *loop:variable* that immediately follows the *symbol* FOR and gives it the current value of the first *numeric:formula* following the = *sign*. This *loop:variable* is then active and may be used as an *integer:variable* until the end of the *independent:statement* that is the latter part of the *loop:statement*. The *loop:variable* is active and may be used in the *formulas* of the other *for:clauses* of the string following the one that activated it. It may even be used in the one or two *formulas* following the *formula* that provides its initial value in the same *for:clause* that activates it. A *for:clause* may be used to activate only a *loop:variable* that is not already active. A given *loop:variable* may be activated by more than one *for: :clause*, but these *for:clauses* must be parts of disjunct *loop:statements*--they must not be included in the same string of *for:clauses* and one must not be nested under another. They are then considered different *loop:variables* in the different *loop:statements*.

A *loop:variable* is activated only by execution of the *for:clause* and remains active only so long as execution remains within the *loop:statement*, except for the cases noted in the next paragraph. A *loop:statement* must not be entered from outside by means of a *go:to:statement* leading (directly or through *switches*) to a *statement:name* inside the *loop:statement*.

This prohibition applies to *statement:names*, *switch:names*, or *close:names* on or within the *independent:statement* forming the latter part of the *loop:statement*. Transfer of control to *statements*, *switch:declarations*, and *close:declarations* within a *loop:statement* from other points within the same *loop:statement* is permitted.

In general the *loop:variables* are deactivated whenever control is transferred outside the loop by means of a *go:to:statement* or by coming out at the bottom of the loop because of completion of the *loop:statement*. The *loop:variables* are not deactivated if control is transferred to a *procedure:declaration*, a *function:declaration*, or a *program:name*; provided the *procedure*, *function*, or outside *program* returns control to the *loop:statement* through the normal exit of the *procedure*, *function*, or *program*.

3.7.6 *PROCESSING:DECLARATIONS* WITHIN *LOOP:STATEMENTS*

Procedure:declarations and *function:declarations* written within a *loop:statement* are not, in any way, associated with the *loop:variables* defined for the *loop: :statement*. The same *loop:variables* may be defined for *loop:statements* within the *procedure:* or *function:declarations* and may be used inside the *procedure:* or *function:declarations* only within such *loop:statements*. Execution of a *procedure:* or *function:declaration* may be invoked from inside or outside any *loop:statement* within which the *declaration* may be written.

On the other hand, *loop:variables* are defined within *switch:declarations* and *close:declarations* that are written as parts of the *loop:statement* for which the *loop:variables* are defined. These *loop:variables* may be used inside such *switch:declarations* and *close:declarations*; these *switch:declarations* and *close:declarations* may be invoked from inside the *loop:statement* in which they occur, but not from outside; and any such *close:declaration* must not contain a *for:clause* defining one of these same *loop:variables*.

3.7.7 ITERATION CONTROL

The compiled instructions needed to do the testing and incrementing specified by *complete:for:clauses* and *two:factor:for:clauses* are inserted at the end of the *loop:statement*. Incrementation of the *loop:variables*, by the current values of the corresponding incrementation *formulas*, takes place in the reverse order of that in which they are defined. If there is no *complete: :for:clause*, incrementation is terminated by an unconditional transfer to the top of the loop, just following initialization of the last *loop:variable*. If the *for:clause* string contains a *complete:for:clause*, incrementation is followed by a test of the controlling *loop:variable*, the one defined by the *complete:for:clause*. If the controlling *loop:variable* has not gone beyond the current value of its limit, control is transferred to the top of the loop; otherwise, execution proceeds to the instructions following the *loop: :statement*.

As mentioned before (section 3.5.4), *test:statement* means one of the two following expressions:

1. TEST θ $
2. TEST θ *loop:variable* θ $

A *test:statement* may only appear within a *loop:statement*. It serves to transfer control to the iteration control routine at the end of a *loop:statement*. Since the iteration control routine is an implied function without a *name*, a *go:to: :statement* cannot be used to transfer control to it. A *test:statement* without a *loop:variable* transfers control to the beginning of the next following iteration control routine for an active *loop:variable*.

A *test:statement* containing a *loop:variable* may only appear in a *loop:statement* in which the referenced *loop:variable* is defined. It serves to transfer control to the point at which the referenced *loop:variable* is incremented. Thus it causes incrementing of the referenced *loop:variable* and all those that precede it in the initialization sequence for the *loop:statement*. If the referenced *loop:variable* is one that is defined by a *one:factor:for:clause*, control is nevertheless transferred to the proper place so that incrementation and testing take place for those *loop:variables* defined in the *loop:statement* before the referenced *loop:variable*.

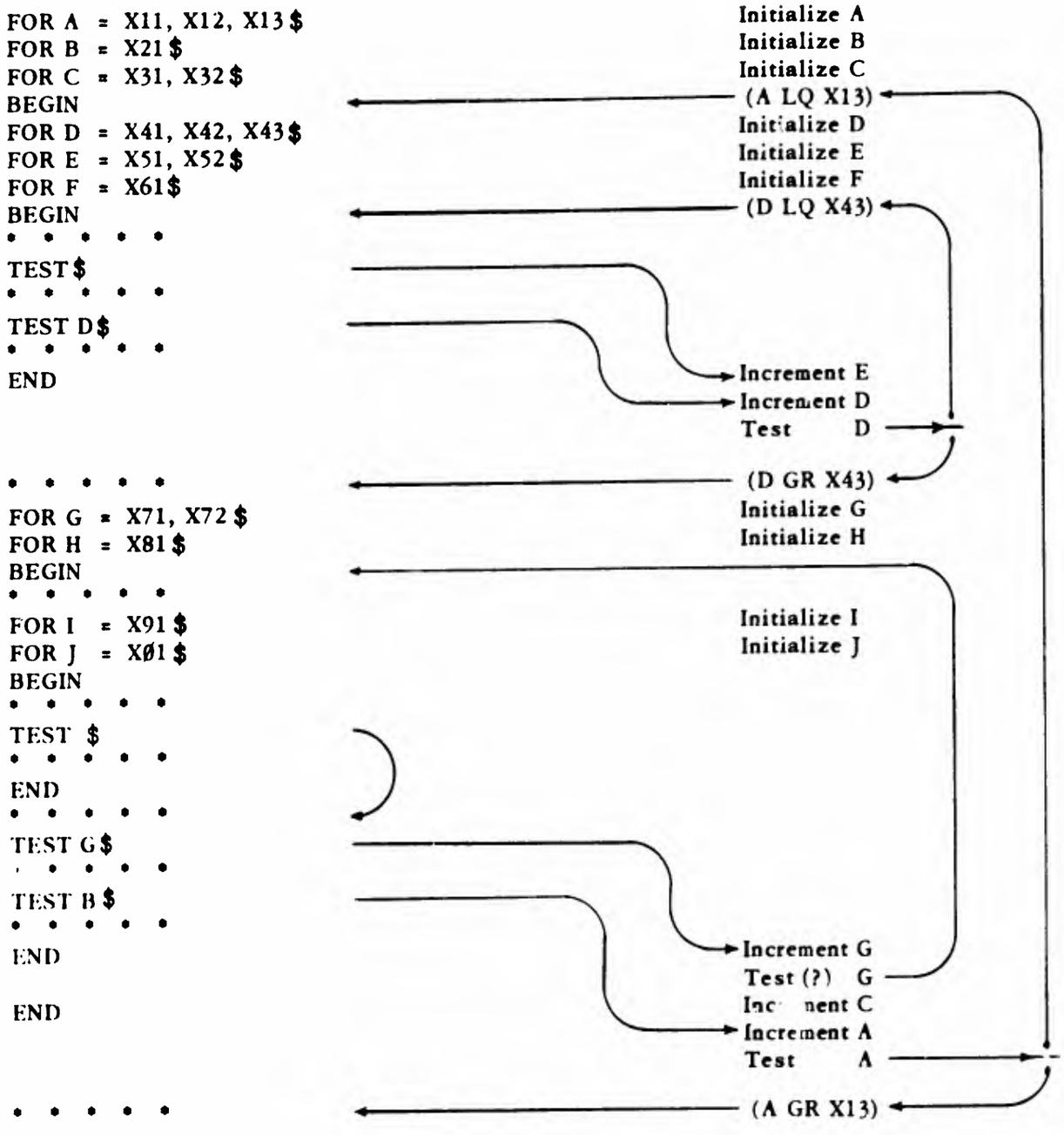


Figure 1. *Loop:Statement* Summary Example

CHAPTER 4. *DECLARATIONS*

4.1 UNDECLARED *NAMES*

Declarations are the principal means of associating *names* with the elements of a *program* or of its environment. This discussion begins by considering the exceptions. A *statement:name* is defined by its appearance followed by a *period*. It is thereby defined as the *name* of the point in the *program* that is the beginning of the next *statement* or *clause*.

4.2 PREDEFINED *NAMES*

Names may be predefined for a *program* as *names* of *items*, *tables*, external *programs*, *procedure:declarations*, or *function:declarations*. Such predefinition is accomplished by means of a compool or a library or both.

A compool (communications pool) is a table or dictionary of definitions for use by a system of related *programs*. If a *program* is to be integrated into the system, the descriptions and locations of common data, *procedures*, and *programs* are found in the compool. A library does not contain descriptions, but rather complete *procedures* or *functions*. If a program calls one of these *procedures* or *functions*, it is copied from the library and effectively made a part of the *program*.

If a *program* written in JOVIAL makes reference to a *name* defined in the compool or library and if this reference is compatible with the compool or library definition, then the reference is taken to be a reference to the compool- or library-defined *name*. If, however, the *program* properly defines such a *name* explicitly then, if there is a conflict, this definition takes precedence and the compool or library definition is disregarded. "Proper" definition has reference to the necessity of placing *program: :declarations* and *data:declarations* ahead of any references to them.

4.4 *DATA:DECLARATIONS*

Data:declarations serve to declare and describe the data on which a *program* is to operate--the inputs, the initial elements of information, the intermediate results, the final results, and the outputs. The *names* given to the data follow the *primitives* that begin the *declarations*, are chosen at the arbitrary discretion of the programmer (or programming supervisor), and have no necessary connection with names used in the outside world--on input manuscripts or printed output, for instance. *Data:declaration* may be sub-

divided into groups as follows:

1. *item:declaration*
2. *table:declaration*
3. *overlay:declaration*

Item:declaration may be further subdivided into the following groups:

4. *simple:item:declaration*
5. *indexed:item:declaration*

And *indexed:item:declaration* may also be called *table:item:declaration*.

Numbers, which are defined in Section 2.6.2, are used extensively as *symbols* in *data:declarations*. In the expressions discussed below, there are several *numbers* in a single form, each with a different meaning. In order to facilitate the explanations, each of the expressions in the following list is defined to be a *number:*

6. *n1n*
7. *n2n*
8. *n3n*
9. *n4n*

The above list is to be understood to be extended, as far as required, in the obvious way. Each of these special ways of writing *number* is used with only one significance in the explanations to follow:

4.4.1 *ITEM:DESCRIPTIONS*

Item:descriptions are parts of *item:declarations* that give the characteristics of the *item*. The adjectives, defined in Section 2.6.1, that apply to *constants* and *variables* also apply to *items*, *item:descriptions*, and *item:declarations*.

Floating:item:description means the *letter* F.

Integer:item:description means one of the expressions in the following list:

1. I θ *n7n* θ S
2. I θ *n7n* θ U
3. A θ *n7n* θ S
4. A θ *n7n* θ U

N7n declares the number of bits required by the *item*, including any sign bit; S declares a signed *item*; U declares an unsigned (positive) *item*.

A, meaning "arithmetic," may be used instead of I in an *integer:item:description.*

Hollerith:item:description means H θ *n7n*

where *n7n* declares the number of bytes in the *item.*

Status:item:declaration means the *abbreviation* S followed by an *optional:n7n* followed by a string of *status:constants.* If present, *n7n* declares the number of bits to be allocated to the *item.* If the given number of bits is k, the number of *status:constants* must not exceed 2^k. If *n7n* is not given, k is determined such that the number of *status:constants* is greater than 2^{k-1} and less than or equal to 2^k. The string of *status:constants* declares all the possible values of the *item.*

4.4.2 *SIMPLE:ITEMS*

Simple:item:declaration means one of the expressions in the following list:

```
1.      ITEM   name  θ  item:description  θ  $
2.      ITEM   name  θ  item:description  θ P θ  optionally:signed:constant
        θ  $
```

The *simple:item:declaration* defines an *item* by naming it and describing it. The second form above also gives it an initial value, the value of the *constant.* The P stands for "preset." The *constant* must be consistent with the *item:description;* that is, it must be of a type that can be assigned to this *item* in an *assignment:statement.* In a *literal:item:declaration* the *constant*, if there is one, must be the same size as the *item.*

Examples of *simple:item:declarations:*

```
3.      ITEM ALPHA  F  $
4.      ITEM MSG H 4 P 4H(HELP)   $
5.      ITEM X2 I 6 S  $
6.      ITEM X3 A 15 U $
```

4.4.3 *INDEPENDENT:OVERLAYS*

Ordinarily the programmer is not concerned with the allocation of the machine registers for the data in his *program,* but it is possible to exert some control over the allocation and for this purpose it is necessary to undstand the conventions assumed by the compiler. The sizes of the various units are system-dependent, usually (but not necessarily) dictated by the computing hardware. In most systems there are some of these units

that are of the same size. In the following discussion it is convenient to relate some units to hardware, but it must be remembered that they are actually chosen arbitrarily by the system designer.

Computer addresses in the main memory are consecutive numbers applied to consecutive groups of ν bits. Each such groups is a "basic addressable unit." For the purpose of allocating space for *simple:items* and *tables*, a unit called a "word" is established consisting of one or more (depending on the system) basic addressable units. A word contains δ bits. A word always contains an integral number of basic addressable units. A "byte" is the unit for encoding a single *sign* or other character of a *literal:item*. A byte consists of ω bits. The maximum number of bytes in a *literal:item* is β. The number of bits in a *floating:item* is η. In any system, the number of bits in a *floating:item* is invariant. In other *numeric:items* the maximum number of bits is π. These sizes, along with others of interest are summarized in Table I (Section 1.2).

Simple:literal:items are allocated from 1 through γ consecutive computer words, ψ bytes per word, depending on the size of the *item*. Each byte is completely contained within one word. If δ is greater than $\psi\times\omega$, then $\delta-\psi\times\omega$ bits of each word (position is system-dependent) are omitted from the allocation to *simple:literal:items*. Except for such omitted bits, any such *item* that is at least ψ bytes long is left-justified in the allocated space. Any such *item* that is less than ψ bytes long is right-justified in its allocated word. Values are right-justified in the *item*; that is, values that are too short are prefaced with *blanks* or zeros (depending on the source) and values that are too long lose bytes from the left by truncation.

All other *simple:items* are allocated one or more (depending on maximum allowable size) computer word of δ bits, each. Thus, *simple:items* are not packed. The allocation of space to *tables* is explained in Section 4.5.5. For the purposes of this section it is sufficient to note that *tables* may begin with a control word containing the length of the *table* in terms of *entries*. When specifying *data:sequences* it is necessary to remember that if this system-dependent control word is present it is the first word of the *table*. Depending on the type of *item* or the type (*floating*, *integer*, etc.) contained in a *table*, it may be that in some systems certain *items* and *tables* may only occupy or begin in certain words.

Subject to the above restrictions, it is possible to specify that storage for *simple:items* and *tables* be allocated in particular sequences. This would not be useful except that it is also possible to specify that these sequences start in the same word. Thus an *item* may have more than one *name*, each *name* corresponding to an entirely different *description* of the *item*. It is even possible for a *table* to overlay several *simple:items*.

Independent:data:sequence means one of the three expressions in the following list:

1. *simple:name:item*
2. *table:name*
3. *independent:data:sequence* θ , θ *independent:data:sequence*

Independent:overlay:specification means one of the expressions in the following list:

4. *independent:data:sequence*
5. *independent:overlay:specification* θ = θ *independent:data:sequence*

Independent:overlay:declaration means one of the three expressions in the following list:

6. OVERLAY θ *independent:overlay:specification* θ $
7. OVERLAY θ *number* θ = θ *independent:overlay:specification* θ $
8. OVERLAY θ *octal:constant* θ = θ *independent:overlay:specification* θ $

An *independent:overlay:declaration* may be used to arrange *simple:items* and *tables* in sequence; to overlay these *sequences* on one another; and to assign these overlays to specific locations. The locations specified, however, must meet the requirements with respect to permitted basic addressable units for all data structures and with respect to the type of *items* involved. Within the *overlay:declaration*, data structures separated by *commas* are given sequential locations in the order in which they are named and *sequences* separated by *equal:signs* begin at the same location. If the *overlay:declaration* contains a *number* or an *octal:constant*, the *constant* gives the number of the basic addressable unit at which the *sequences* are to begin. Otherwise, the common origin is selected by the compiler so that it will not conflict with other data or program storage. Examples:

9. OVERLAY LIST = DUMMY, MESSAGE $
10. OVERLAY 1Ø24 = UMPIRE $

The *name* of a data structure may appear no more than once in *overlay:declarations*. Data structures named in an *overlay:declaration* must first be defined; either predefined by compool, or declared previously in the *program*. Compool-defined *items* and *tables*, if they appear in *overlay:declarations*, must precede all other *names*. Data structures appearing in *overlay:declarations* must not be provided with initial values.

4.5 COMPLEX DATA STRUCTURES

It is often necessary to specify more complex data structures than *simple:items*. *Tables* serve this need. A *table* is basically a one-dimensional arrangement (or list) of *entries*, the particular *entry* being designated by a one-component *index*. Each *entry* is a group of *items*, each having a unique *item:name*. For example ALPHA ($ 5 $) might be one of several *items* in *entry* 5 of a particular *table*, or it might be the only *item* in *entry* 5.

4.5.1 *CONSTANT:LISTS*

It is sometimes desirable to specify initial values for all or part of a *table* when it is declared. Such initial values are specified in lists known as *constant:lists*. A *one:dimensional:constant:list* is the *primitive* BEGIN followed by a string of *optionally:signed:constants* followed by the *primitive* END. Example:

```
1.      BEGIN  -13.  78.  35.  -16.  Ø.  64.  END
```

4.5.4 TYPE MATCHING FOR PRESET VALUES

As with a single *constant* given as initial value of a *simple:item*, each of the *constants* in a *constant:list* must be of a type that can be assigned to the associated *item*. Furthermore, certain mixtures of type are allowed, while others are not. The permissible type mixtures of the *constants* given as initial values of *indexed:items* are shown in Table V. Example of a mixed *constant:list*:

```
1.      BEGIN  Ø  Ø  1.2  -1  END
```

Table V. *Constant* Types for Presetting *Items*

Item type	*Constant* types
Integer or *floating*	*Integer*, *floating*, or mixtures of these types
Integer or *floating*	*Octal*
Hollerith	*Hollerith* or *octal*, but not mixed
Status	*Status*

4.5.5 *TABLES*

In the structure of a *table*, variously described parts of the *table* have distinct *names*. These separately named parts of the *table* must be declared within the *table:declaration*.

A *table* may have a rigid or variable number of *entries*. Dependent on the system, a *table* may begin with a δ-bit control word containg the number of *entries*. Regardless of the system, however, the number of *entries* is accessible through the use of NENT. Within any system, either all *tables* begin with control words or none do.

There are two kinds of *table:declarations* as follows:

1. *ordinary:table:declaration*
2. *defined:entry:table:declaration*

There are subordinate *declarations* that can be used only within *table: :declarations*. These subordinate *declarations* are explained in Sections 4.5.6 and 4.5.8.

The structure of an *entry* of a *table* is described in terms of *items*. These *items* may be arranged without packing, in which case computer space is allocated as for *simple:items* (see Section 4.4.3), except for the differences due to parallel structure (see Section 4.5.7).

In *tables* with medium packing, space is allocated in units of bytes, as many bytes as required. If $\psi \times \omega$ is less than δ, then $\delta - \psi \times \omega$ (system-dependent) bits of each word are not used for *literal:items*, and *items*, other than *literal:items*, that don't fit in ψ bytes remain unpacked in *tables* with medium packing.

Within the bytes allocated to a medium-packed *item*, representation and bit numbering is the same as for a *simple:item*, but scaled down to the allocated space. If ω equals 6, a nine-bit, signed *integer:item* is medium packed in two bytes; the sign is the first bit of the first byte; and there are three filler bits between the sign (bit ∅) and bit 1. The storing and getting of values of medium-packed *items* is done in terms of the allocated bytes; adjacent bytes are not picked up when getting such an *item* and not disturbed when setting it.

In *tables* with dense packing, space is allocated to *literal:items* as in *tables* with medium packing, but to other *items* it is allocated by bit. Representation, bit numbering, storing, and getting of densely packed *items* is done with reference to the allocated bits; adjacent bits are neither picked up nor disturbed when getting or setting such an item.

Literal:items in packed *tables* are allocated consecutive bytes in the *entry*, skipping only the system-dependent $\delta - \psi \times \omega$ bits of each word. Packed *items* are always assigned space so that each is completely contained in as few words as possible. The *items* of a packed *table* are allocated space in an order that most efficiently packs them, subject to the above rules.

In *defined:entry:tables* (see Sections 4.5.8 and 4.5.9) the allocation of storage is explicitly specified by the programmer. However, he may indicate the effect of his allocation in terms of the type of packing that results, as it applies to each *item*. If he doesn't specify this, the *item* is assumed to be densely packed. Representation and bit numbering follow the rules for dense packing in any case, but if the *item* is stated not to be packed or to be medium packed, the appropriate word or bytes are referenced for setting and getting the value. If a signed *item* is stated not to be packed or to be medium packed in contradiction of the fact, setting or getting the *item* leads to erroneous results.

4.5.6 *ORDINARY:ENTRIES*

Ordinary:table:item:declaration means one of the two following expressions:

1. ITEM θ *name* θ *item:description* θ $
2. ITEM θ *name* θ *item:description* θ $ θ *one:dimensional:constant:list*

The permissible *item:descriptions* are the same, and have the same meanings, in *simple:item:declarations* and all other *item:declarations*. (Sections 4.4.1 and 4.4.2). The *ordinary:table:item:declaration* declares, names, and describes an *item* for every *entry* of the *table* with which it is declared (as explained below). In referring to a particular *item* in a particular *entry* the *item:name* and a one-component *index* are used as in the following examples (indexing the *entries* of a *table* begins with zero):

```
3.
4.      ALPHA       (S Ø $)
        BETA        ($ I + 5 $)
5.      MESSAG      ($ ALPHA ($ K $) * 2 $)
```

It may be that a particular *item* is not present in a particular *entry* of the *table*, but is present in subsequent *entries*. For instance, an *overlay:declaration* (as explained below) may be part of the *table:declaration*. The compiler has no way, in general, of knowing which *entries* contain which *items*, since this is determined by usage rather than *declarations*. In any case such information is ignored and indexing is accomplished as if every *entry* contained every declared *item*.

A *table:item:declaration* containing a *one:dimensional:constant:list* specifies initial values for the *item* in *entries* of the *table*. The first value is given to the *item* in *entry* Ø, the second value to the *item* in *entry* 1, and so forth. No cognizance is taken that in actual usage this *item* might not exist in a particular *entry*. The number of *constants* in the *constant:list* may be less than the number of *entries* specified for the *table* (Section 4.5.7), but there must not be too many *constants* for the number of *entries*. If there are not enough *constants* to complete the initial assignments, no values are provided for the *item* in the remaining *entries* at the end of the *table*. The specific values of *items* for which no initial values have been provided are undefined at the start of execution.

Subordinate:overlay:specification means one of these two expressions:

6. *ordinary:table:item:name* θ = θ *ordinary:table:item:name*
7. *ordinary:table:item:name* θ = θ *subordinate:overlay:specification*

Subordinate:overlay:declaration means

8. OVERLAY θ *subordinate:overlay:specification* θ $

Ordinary:entry:description means one of the three following expressions:

9. *ordinary:table:item:declaration*
10. *ordinary:entry:description* θ *ordinary:entry:description*
11. *ordinary:entry:description* θ *subordinate:overlay:declaration*

In other words, an *ordinary:entry:description* is a string of *ordinary:table:item:declarations* and *subordinate:overlay:declarations*. The form is restricted in that all *item:names* appearing in any *overlay:declaration* in the *entry:description* must be previously declared in *item:declarations* occurring earlier in the same *entry:description*. An *ordinary:entry:description* names and describes all the *items* that constitute a *table:entry*.

A *subordinate:overlay:declaration* within the *entry:description* equates *items* of the *entry*. *Items* separated by *equals:signs* are given the same location within the *entry*. Note that a *subordinate:overlay:declaration* cannot specify an absolute location as origin and cannot specify *sequences*.

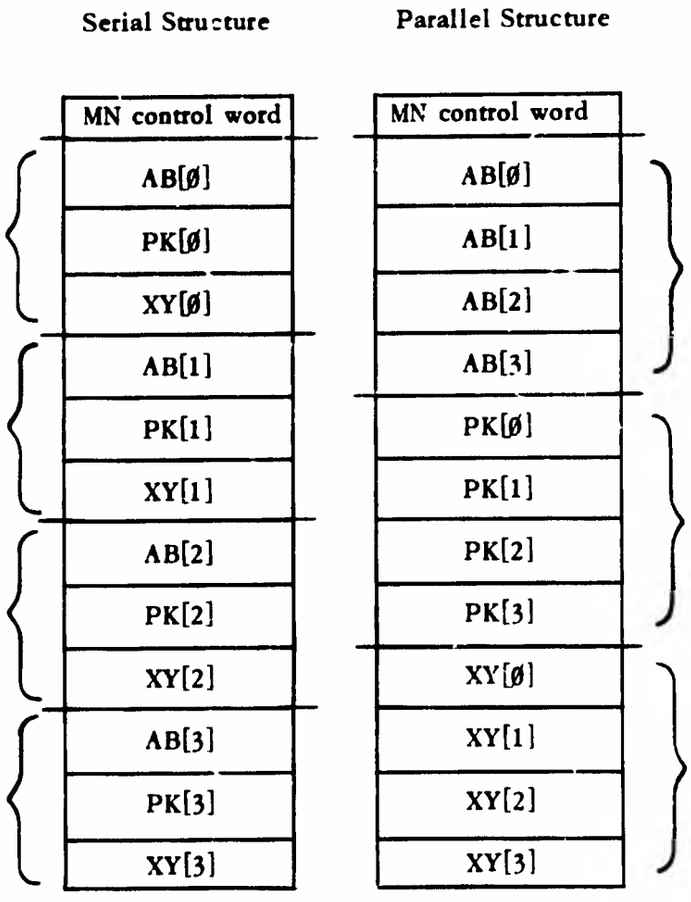
A *name* must not appear in more than one *subordinate:overlay:declaration* within any *ordinary:entry:description*.

4.5.7 *ORDINARY:TABLES*

Table:size:specification means one of the two following expressions:

1. V θ *n1n*
2. R θ *n1n*

A *table:size:specification* declares the size of a *table* in terms of the *number of entries* in the *table*. The V means that the size of the *table* is variable; that *n1n* is the maximum number of *entries* in the *table*; and that NENT (*table:name*) is a *numeric:variable*. The R means that the *table* is of a rigid size with *n1n entries* and that NENT (*table:name*) may be used as a *constant* but not a *variable*. The value of a variable NENT (*table:name*) is undefined prior to being set by an *assignment:statement* or some other means. The value of NENT (*table:name*) is an unsigned *integer* value.



Example: *Table* MN has 3 *items:* AB, PK, XY and 4 *entries:* Ø, 1, 2, and 3.

Thirteen consecutive computer words are shown in each illustration above.

Figure 3. Serial and Parallel *Table* Structure

Basic:structure:specification means the *letter* P or the *letter* S. It is used to specify the basic structural pattern of the *table*, P declaring parallel structure and S declaring serial structure. Parallel and serial structure can best be explained in terms of the sizes of a *table* and its *entries*. From the previous paragraph we have that *n1n* is the (maximum) number of *entries* in the *table*. Let *n2n* be the number of words in an *entry*. In serial *tables* there are *n1n* consecutive blocks of storage, blocks being allocated to *entries* in numerical order, and each block consisting of the *n2n* consecutive words of the *entry*. In parallel *tables* there are *n2n* blocks of *n1n* words, each block containing

one word from each *entry* of the *table*. (See Figures 3 and 4). In some systems, each *table*, regardless of its basic structure, begins with a control word containing the value of NENT.

The use of a *basic:structure:specification* in a *table:declaration* is completely optional. In the absence of a *basic:structure:specification*, the *table* has parallel structure. In *tables* with variable-length *entries* (Section 4.5.9), the basic structure must be declared serial.

Packing:specification means the *letter* N or the *letter* M or the *letter* D. It is used to specify the packing of *items* within an *entry* of a *table*. N stands for "no packing," which means that each *item* occupies its own word or words without sharing it (or them) with other *items*. "No packing" does not preclude "overlaying." D stands for "dense packing," which means that *items* are packed

	MN control word	Serial Structure		
AB	2nd word	5th word	8th word	11th word
PK	3rd word	6th word	9th word	12th word
XY	4th word	7th word	10th word	13th word
	Entry Ø	*Entry* 1	*Entry* 2	*Entry* 3

	MN control word	Parallel Structure	
Entry Ø	2nd word	6th word	10th word
Entry 1	3rd word	7th word	11th word
Entry 2	4th word	8th word	12th word
Entry 3	5th word	9th word	13th word
	AB	PK	XY

This illustrates the same example as Figure 3. The same 13 words are shown, but the blocks are drawn side by side.

Figure 4. Parallel and Serial *Table* Structure

together to utilize every bit in the word and minimize space, except that no *item* is ever packed so as to occupy more words than are required to contain it. M stands for "medium packing," which means that *items* are packed but not so closely that they share bytes. The *packing:specification* may be omitted; this implies "no packing" in *ordinary:entries*.

Ordinary:table:declaration means the following structure:

3. TABLE θ *name* θ *table:size:specification* θ
optional:basic:structure:specification θ
optional:packing:specification θ $ θ
BEGIN θ *ordinary:entry:description* θ END

The *size:specification* tells whether the *table* has a variable or rigid number of *entries* and how many. The *basic:structure:specification*, if present, declares the *table* to be parallel or serial. The *packing:specification*, if present, declares medium or dense packing or none. The *entry:description* declares, names, and describes all the *items* of the *table* and any overlaying among these *items*.

Example:

```
4.      TABLE    TB2     V      1ØØ   P    N    $
        BEGIN    ITEM    ALF      H   2    $
                 BEGIN   2H(PH)   2H(**) 2H(U2)  END
                 ITEM    NUM  I   12  U $
                 OVERLAY     ALF = NUM $
        END
```

4.5.8 *DEFINED:ENTRIES*

Defined:entry:item:declaration means the following expression:

1. ITEM θ *name* θ *item:description* θ *n3n* θ *n4n* θ
optional:packing:specification θ $ θ
optional:one:dimensional:constant:list

The elements of the above expression that are also included in the *ordinary::table:item:declaration* (Section 4.5.6) serve the same purpose here that they serve there. This *declaration* occurs only within a *table:declaration* (Section 4.5.9) in which the number of words in an *entry* is specified. In this *declaration*, *n3n* declares which word in the *entry* contains the *item* (or in which word the *item* begins). For this purpose the words are numbered starting with Ø. Thus, the words of a four-word *entry* are numbered Ø, 1, 2, and 3. The bit within the word in which the *item* begins is specified by *n4n*. The bits are numbered from the left starting with Ø. The *item:description* and the use of the *optional:one:dimensional:constant:list* to set initial values are explained in Section 4.5.6. The *packing:specification* may be used to

provide information useful to the compiler. It does not direct the packing of the *item*, but describes the packing that results from *n3n*, *n4n*, the *item::description*, and the situation of adjacent *items* in the *entry*. In *defined::entries*, omission of the *packing:specification* implies "dense packing."

Defined:entry:description means one of the following two expressions:

2. *defined:entry:item:declaration*
3. *defined:entry:description* θ *defined:entry:description*

In other words, a *defined:entry:description* is a string of *defined:entry::item:declarations*. Notice that *overlay:declarations* are not permitted in a *defined:entry:description*. They are not needed since the position in the *entry* of every *item* is explicitly declared, including any desired overlaying or partial overlaying.

4.5.9 *DEFINED:ENTRY:TABLES*

Defined:entry:table:declaration means the following expression:

1. TABLE θ *name* θ *table:size:specification* θ
   optional:basic:structure:specification θ *n2n* θ $ θ
   BEGIN θ *defined:entry:description* θ END

The *size:specification* indicates a variable or rigid number of *entries* and how many. The *basic:structure:specification*, if present, declares parallel or serial *table* structure. *N2n* declares the number of words per *entry*. The *entry:description* declares, names, and describes all the *items* of the *table* and defines their positions within the *entries*. A *literal:item* that crosses a word boundary must start with the first bit of some byte as bytes are defined for the system. (See Section 4.5.5). The compiler then omits the $\delta - \psi \times \omega$ system-dependent bits of each word of the *entry* used for allocation of the subsequent bytes of the *item*, but allocates bytes in order from the left. Any *item* other than a *literal:item* may be declared to start in any bit of a word so long as this does not cause it to overflow into more words than are needed to contain it.

Notice that *n2n* is now required to state the size of an *entry*. However, *n2n* is really only a nominal or assumed *entry* size. The compiler uses *n2n* (and, of course, *n1n*, the number of *entries*) to allocate space for the *table*--*n2n* blocks of *n1n* words or *n1n* blocks of *n2n* words depending on the basic structure. See, for example, Figure 4.

It is permissible to declare a *table:item* to begin in a word of the *entry* that, by *n2n*, doesn't exist. That is, *n3n* may be equal to or greater than *n2n*. The *table* structure is used to find the beginning of a referenced *entry* and *n3n* is used to find the *item*. A *table* declared and used in this way is considered to have variable-length *entries*.

For *tables* with variable length *entries* the compiler takes no extra pains beyond what has just been described. Therefore, it is up to the programmer to be aware of the differences between his conception of the *table* and the way the compiler treats it and to write his *program* accordingly.

Among the things that must be considered:

- The *table* must be of serial *entry* structure. This is so even if there is only one word per *entry*. Consideration of Figure 4 shows that for a serial *table*, for all *entries* except the last, a reference beyond the end of an *entry* spills over into the next *entry*. For a parallel *table*, on the other hand, a reference beyond the end of an *entry* is a reference completely outside the block allocated to the *table*.

- In assigning preset values and in interpreting *indices*, every declared *item* is considered to be associated with every *entry* of the *table* regardless of conflicts due to overlays. For example, you may know that there is no ALPHA in *entry* 7 because GAMMA ($ 6 $) actually occupies that space, but the compiler doesn't know it. When referring to the next ALPHA that does exist, call it ALPHA ($ 8 $) and not ALPHA ($ 7 $).

- The programmer must provide for any extra incrementing that may be necessary while indexing through a *table* by means of a *loop: :statement*. For instance, some such coding may be required as below:

```
FOR Q = Ø, 1, NENT (XXX) -1 $
BEGIN
    :
    :
IF SIZE ($ Q $) EQ 2 $
Q = Q + 1 $
END
```

- It is probably necessary that the nominal *entry* size, *n2n*, be a divisor of each of the various actual *entry* sizes that the programmer has in mind for the particular *table*. If each of these "actual" sizes is not a multiple of *n2n*, there will be wasted space in the *table*, or the required programming adjustments will be impossible, or both. Of course, one way of satisfying this requirement is to use the value 1 for *n2n*.

CHAPTER 5. *PROCESSING:DECLARATIONS*

5.1 INTRODUCTION

Unless otherwise directed (Section 6 1) every *program* begins execution with the first *statement* of the *main:program*. *Statement* execution then proceeds sequentially except for iterations of *loop:statements* and jumps due to *go:to:statements* and *conditional:statements*. In (almost) every *program*, however, there are groupings of *statements* or other elements of the *program* to which execution control cannot pass sequentially, but only through invocation of the group or element by *name*. Such groups or elements are defined as *processing:declarations*.

A *statement* or group of *statements* that is blocked from sequential access only because of the presence of *go:to:statements* or *conditional: :statements* is not thereby a *processing:declaration*.

The following list enumerates all the *processing:declarations:*

1. *program:declaration*
2. *switch:declaration*
3. *close:declaration*
4. *procedure:declaration*
5. *function:declaration*

5.2 *PROGRAM:DECLARATIONS*

Origin means one of the following expressions:

1. *number*
2. *octal:constant*

Program declaration means:

3. 'PROGRAM θ *name* θ *optional:origin* θ $

Notice that the *primitive* introducing the above expression is spelled with a leading *prime*. A *program:declaration* serves to establish communication between the present *program* and another *program*, named in the *declaration* and compiled independently as a closed subroutine. If present, the *origin* declares the machine address (number of the basic addressable unit) of the beginning compiled location of the named *program*. The presence or absence of the *origin* is system-dependent. If the system supplies the machine

location and if it is not desired or not permitted to override this given location, the *origin* is omitted. When transfer to the named *program* is specified by means of a *go:to:statement*, the compiler assumes that the named *program* is a subroutine, which returns control to the *statement* following the *go:to:statement*; and that the values of any *loop:variables* that are active at the execution of the *go:to:statement* are undisturbed upon return from the subroutine.

A *program:declaration* is a *processing:declaration* since it names a group of *statements* to which control can be transferred. However, it shares with *data:declarations* the property of not directly generating any machine language coding; it can occur among the *statements* of a *program* without affecting the order of execution.

5.3 *SWITCHES*

A *switch:declaration* includes a list of *statement:names* that specify points to which execution control may be transferred, depending on the value of an *item* or an *index*. *Switch:declaration* is therefore divided into the following categories:

1. *index:switch:declaration*
2. *item:switch:declaration*

A *switch:declaration* causes the generation of machine language instructions that are executed only when the *switch:name* is invoked by a *go:to:statement*. A *switch:declaration* must not be placed in such a position that the execution sequence can fall into it.

A *switch:declaration* within a *loop:statement* must not be invoked by a *go:to:statement* outside that *loop:statement*. For more details see Section 3.7.6 on *processing:declarations* within *loop:statements*.

5.3.1 *INDEX:SWITCHES*

Index:switch:list means one of the following expressions:

1. *statement:name*
2. , θ *statement:name*
3. *statement:name* θ ,
4. *index:switch:list* θ , θ *index:switch:list*

Index:switch:declaration means:

5. SWITCH θ *name* θ = θ (θ *index:switch:list* θ) θ $

The *name* in the above expression is thereby declared to be the *switch:name*. An example of an *index:switch:declaration* is presented below:

```
6.      SWITCH TOGGLE = (BL97, , LOOP , EMIT , ) $
```

To invoke an *index:switch*, the *switch:name* with a one-component *index* is the *sequence:designator* in a *go:to:statement*. For example:

```
7.      GOTO TOGGLE ($ Ø $) $
8.      GOTO TOGGLE ($ K $) $
```

The *index* value points out the required *statement:name* according to its position in the list, starting with zero. *Commas* without corresponding *statement:names* indicate values of the *index* for which no transfer of control takes place. The *index* in a reference to an *index:switch* must be within the range indicated by the *switch:list* in the *declaration*. Thus, GOTO TOGGLE ($ Ø $) $ effects a transfer to BL97. If the reference to TOGGLE ($ K $) is activated; for K = 2, control is transferred to LOOP; for K = 3, control is transferred to EMIT; for K = 1 or 4, control is not transferred, but is returned to the *statement* following the invoking *go:to:statement*; and K must not be more than 4.

5.3.2 *ITEM:SWITCHES*

Item:switch:list means one of the two following expressions:

```
1.      constant θ = θ statement:name
2.      item:switch:list θ , θ item:switch:list
```

Item:switch:declaration means the following expression:

```
3.      SWITCH θ name θ (θ item:name θ) θ = θ (θ item:switch:list θ) θ $
```

The *name* following the *primitive*, SWITCH, is the *switch:name*. The *item:switch:list* consists of *constants* paired with *statement:names*. The *constants* are possible values of the *item* named in the *declaration*. When the *switch* is invoked, if the value of the *item* matches one of the *constants*, execution control is transferred to the corresponding *statement:name*. If the *item* value doesn't match any of the *constants*, execution continues with the *statement* following the invoking *go:to:statement*.

Example of an *item:switch:declaration*:

```
4.      SWITCH  WHICH (BETA) = (3H(ARY) = ST34, 3H(ØL9) = FINIS,
        3H(   ) = SØ1, 3H(ABC) = SØ2, 3H(''') = EXIT, 3H(===) = SØ1,
        3H(.$.) = ESSO, O(777777) = STØ1, 3H(XXX) = PCR ) $
```

If the *switch:declaration* names a *simple:item*, any reference to the *switch* omits an *index*. If the *switch:declaration* names an *indexed:item*, reference to the *switch* includes an *index* to select the particular *entry* of the *item* to be compared with the *constants*. For example:

```
5.        GOTO  WHICH    ($  J  $)  $
```

This *go:to:statement* implies that BETA (the *item* named in the *declaration* for WHICH) is an *indexed:item*.

5.4 *CLOSES*

Close:declaration means the following expression:

```
1.        CLOSE  θ  name  θ  $  θ  BEGIN  θ  statement:list  θ  END
```

Close:declarations, as well as *procedure:* and *function:declarations*, provide the means for setting up groups of *statements* as subroutines to be called upon or invoked from various points in a *program*. The *statement:list* must not contain any *close:declarations*; thus, *close:declarations* cannot be nested. A *close:declaration* may invoke *procedures* or *functions* or other *close:declarations*, but there must be no recursive calls. That is, no subroutine may call itself nor any other subroutine that in turn calls it, either directly or indirectly. The *name* in the above *declaration* becomes the *close:name*.

A *close:declaration* sets up the statement forming its latter part as a closed subroutine without parameters. A *close:declaration* must not be placed in such a position that the execution sequence can fall into it. The processing specified by a *close:declaration* is executed when the *close:name* is invoked by a *go:to:statement*. Normally, after execution of a *close:declaration*, control returns to the *statement* following the invoking *go:to:statement*. It is permissible, however, for there to be a *go:to:statement*, within the *close:declaration*, that jumps to an entirely independent point in the *program*.

A *close:declaration* within a *loop:statement* must not be called by a *go:to:statement* outside that *loop:statement*. A *close:declaration* outside the *loop:statement* should be invoked from within the *loop:statement* only if the *close:declaration* does not return control to the *loop:statement*. See Section 3.7.6 on *processing:declarations* within *loop:statements* for more details. A *close:declaration* must not be invoked from within a *procedure::body* unless the *close:declaration* is within the *procedure:body*.

5.5 *PROCEDURES*

A *procedure:declaration* sets up a closed subroutine that may have *input: :parameters* or *output:parameters* or both. A *procedure:declaration* is independent of outside *loop:statements;* it may be invoked from within any *loop:statement* in the *main:program* or in other *processing:declarations* without deactivating the *loop:variables.* On the other hand, the outside *loop:variables* are not defined in the *procedure:declaration.*

Some preliminary definitions are needed.

Declaration:list means one of the following expressions:

1. *directive*
2. *data:declaration*
3. *program:declaration*
4. *declaration:list* θ *declaration:list*

Formal:input:parameter:list means one of the two following expressions:

5. *simple:item:name*
6. *formal:input:parameter:list* θ , θ *simple:item:name*

Formal:output:parameter:list means one of the two following expressions:

7. *simple:item:name*
8. *formal:output:parameter:list* θ , θ *simple:item:name*

Procedure:heading means one of the following three expressions:

9. PROC θ *name* θ $ θ *optional:declaration:list*
10. PROC θ *name* θ (θ *optional:formal:input:parameter:list* θ) θ $ θ *optional:declaration:list*
11. PROC θ *name* θ (θ *optional:formal:input:parameter:list* θ = θ *formal:output:parameter:list* θ) θ $ θ *optional:declaration:list*

Procedure:body means:

12. BEGIN θ *statement:list* θ END

Procedure:declaration means:

13. *procedure:heading* θ *procedure:body*

The *statement:list* of a *procedure:body* is restricted in that it must not contain any *procedure:declarations* nor *function:declarations.* Thus, *procedure:declarations* cannot be nested, although it is permissible for a

procedure:declaration to contain *procedure:call:statements* or *function:calls*. There must not be any recursive calls, however. That is, a *procedure* must not call itself nor any *close*, *procedure*, or *function* that calls it in turn, either directly or indirectly.

The *name* following PROC becomes the *procedure:name*. A *procedure:declaration* sets up a closed subroutine (or *procedure*) that is invoked by a *procedure: :call:statement*. A *procedure:declaration* must not be placed in such a position that the execution sequence can fall into it. Normally, when execution of the *statement:list* is completed or a *return:statement* is executed, control returns to the *statement* that follows the invoking *procedure:call:statement*. As control passes from the *procedure*, *actual:output:parameters* corresponding to *simple:item:names* in the *formal:output:parameter:list* are assigned the values calculated by the *procedure*, in order from left to right as if specified by a sequence of *assignment:statements*. It is possible, however, for the *procedure* to contain a *go:to:statement* or *switch* that references a *name* in the *main:program*. If control passes to the *main:program* through execution of such a *go:to:statement* or *switch*, then the final assignment process is bypassed and the *actual:output:parameters* are not changed. It is also possible that *loop:variables* in the *main:program* that are active at the time of calling the *procedure*, do not have their correct values. See Section 3.5.5 for more details on the usage of *input:parameters* and *output:parameters*.

5.6 *FUNCTION:DECLARATIONS*

A *function:declaration* is very similar to a *procedure:declaration;* so much so in fact that the same *primitive*, PROC, is used to introduce both.

Function:heading means one of the following expressions:

1. PROC θ *name* θ $ θ *optional:declaration:list*
2. PROC θ *name* θ (θ *optional:formal:input:parameter:list* θ) θ $ θ *optional:declaration:list*

Function:declaration means:

3. *function:heading* θ *procedure:body*

A *function:declaration* is distinguished from a *procedure:declaration* by the presence, in a *function:declaration*, of a *simple:item:declaration* declaring an *item* with the same *name* as the *function:declaration*. It is this *item* with the matching *name* that is to carry the value of the *function*. This *item* is to be treated, within the *function:declaration*, as the sole *output:parameter* although the *function:declaration* does not provide for a *formal:output:parameter:list*.

The type of the *item* that acts as the *output:parameter* determines the type of *formula* represented by a *function:call*. The discussion in Section 3.5.5 concerning *input:parameters* applies to *function:declarations* and corresponding *function:calls*.

Function:declarations may contain *procedure:call:statements* or *function: :calls*, but not recursively. *Function:declarations* must not contain *function:declarations* nor *procedure:declarations*. A *function:declaration*, just as a *procedure:declaration*, is independent of outside *loop:statements*, and must not be placed in such a position that the execution sequence can fall into it.

All *formal:input:parameters* must be declared in the *function:heading*.

5.7 REM AND REMQUO, BUILT-IN *FUNCTION* AND BUILT-IN *PROCEDURE*

The compiler recognizes the following two predefined *names* regardless of the presence of a compool or library:

1. REM
2. REMQUO

It is, of course, possible to override the predefinitions (see Section 4.2).

REM is a *function* that yields the remainder in a division of two *integers*. It is defined as follows:

3.
```
PROC    REM    (NUM, DEN)$
ITEM    REM       I    6  S  $
ITEM    NUM       I    6  S  $
ITEM    DEN       I    6  S  $
BEGIN   REM =  NUM/DEN  $
        REM =  NUM-REM  *  DEN  $
END
```

REMQUO is a *procedure* that yields both remainder and quotient of a division of two *integers*. It is defined as follows:

4.
```
PROC    REMQUO  (NUM, DEN = REM,QUO)  $
ITEM    NUM    I   6    S   $
ITEM    DEN    I   6    S   $
ITEM    REM    I   6    S   $
ITEM    QUO    I   6    S   $
BEGIN   QUO = NUM/DEN  $
        REM = NUM - QUO * DEN  $
END
```

The definitions are so simple it might be asked why these routines are predefined. The answer has to do with machine characteristics: these particular operations are often required, but the definition in JOVIAL is usually awkward in comparison with a definition in machine language. Consequently, calls on these routines are usually implemented, not as ordinary *functions* and *procedures*, but as highly efficient open (or in-line) machine code. It is the effect that is specified here, not the means for achieving it nor the efficiency.

CHAPTER 6. *PROGRAMS*

6.1 *PROGRAM*

The *program* can be compiled as an independent entity, subject to system requirements concerning an executive or monitor, or as a closed subroutine, which returns control to a calling *program* after execution. The machine location at which the compiled *program* is to begin may be specified in octal or decimal or left up to the compiler. Execution of the compiled *program* begins with the first *statement* of the *main:program* or with a *statement* specified by *name*. The specification of a starting point by *name* is permitted only in an independent *program*. These choices are indicated in the *program*.

Program means one of these two expressions:

1. START θ *optional:origin* θ $ θ *statement:list* θ TERM θ *optional:statement:name* θ $
2. CLOSE θ *name* θ $ θ START θ *optional:origin* θ $ θ *statement:list* θ TERM θ $

In other words a *program* is a string of *statements*, *declarations*, and *directives* (see *statement:list*, Section 3.6) enclosed between certain prefatory and terminating delimiters.

If the *program* begins with CLOSE, it is a closed subroutine and the *name* before START is the *name* of this subroutine. Otherwise this is an independent *program*. If the *origin* is present, it specifies the machine location at which the compiled *program* begins. If there is no *origin* stated, the compiler chooses the starting location. If there is a *statement:name* between TERM and the following *dollar:sign*, it must be the *name* of a *statement* in the *main:program* not within any *loop:statement* or *processing:declaration*; it is the *name* of the *statement* to be executed first. Otherwise execution begins with the first *statement* of the *main:program*.

6.4 SCOPE OF DEFINITION OF *NAMES*

There are over twelve million *names* available to JOVIAL programmers if we consider only those with no more than six *letters* and *numerals*. Nevertheless, programmers seem to concentrate on a very few out of these millions. JOVIAL caters to this tendency by providing for duplication of *names* in accordance with the criteria explained below.

Loop:variables are not *names*; yet the scope of their definitions is of critical importance. This is explained in detail in Sections 3.7.5, 3.7.6, and 3.7.7. In connection with *loop:variables*, "defined" means the same as "active."

In *status:constants* the *names* within *parentheses* have no connection with *names* used elsewhere in the *program*. There need be no concern about duplication except that there must be no duplication among the *status:constants* associated with any particular *status:item*.

If the *program* has a *name* before START, this *name* has no connection with *names* used elsewhere in the *program*. There need be no concern about it possibly being a duplication.

Following a *define:directive*, any occurrence of the *name* thereby defined is effectively replaced by its definition, with these exceptions--the *name* may be redefined by a new *define:directive;* there is no replacement where the *name* occurs as part of a *status:constant*, *literal:constant*, or *comment;* the *name* is replaced where it appears within *direct:assigns* but not elsewhere in *direct:code*.

Let us now consider the *names* in the *program* after all effective replacements in accordance with *define:directives*. There may be duplication of *names* if the elements so named have non-overlapping scope. "Scope" has reference to the setting off of parts of the *program* in *procedure: :declarations* and *function:declarations*. In general, a *name* that is defined in a particular way just within a *procedure:* or *function:declaration* is said to be "local" to that *procedure:* or *function:declaration*. All of the *program* that is <u>not</u> part of a *procedure:* or *function:declaration* is the *main:program*. A *name* that is predefined or defined within the *main:program* is said to be "global."

All *names* may be predefined (by compool or otherwise). All *names* may be explicitly defined--*statement:names* by being properly prefixed to *statements* or *clauses;* the others by *declarations*. Predefined *names* are global. *Names* explicitly defined in the *main:program* are global. If an explicit definition in the *main:program* conflicts with a predefinition, the predefinition is nullified. Conflicting global definitions in the *main:program* are not allowed. *Names* explicitly defined within a *procedure:* or *function: :declaration* are local to that particular *procedure:* or *function:declaration*. This includes all *formal:parameters*. Conflicting local definitions within a particular *procedure:* or *function:declaration* are not allowed. A *name*, local to a given *procedure:declaration*, must not be used as both a *formal:input:parameter* and a *formal:output:parameter*. A *procedure:name* or *function:name* is both global and local to the *procedure:declaration* or *function:declaration* that it names. One seeming conflict in naming is not only permitted but required: a *function:declaration* must contain a *simple:item:declaration* that duplicates the *function:name*.

The scope of a local *name* is the *procedure:* or *function:declaration* in which it is defined. The scope of a global *name* is the *main:program* and all *procedure:* and *function:declarations* that do not have local definitions of the same *name*. Consider Figure 7 for example. Assume that the definitions are made explicitly as shown in the *main:program* or *procedures*. Then, with respect to Figure 7A, T12 is:

1. as a *statement:name*, global; and its scope is the *main:program* and *procedure* P2.
2. as an *item:name*, local to *procedure* P1.

With respect to Figure 7B, T13 is:

3. as a *procedure:name*, both global and local to *procedure* T13; its scope is the *main:program* and *procedure* T13.
4. as a *program:name*, local to *procedure* P3.

Reference to any defined *name*, as in a *go:to:statement*, an *assignment:statement*, or a *procedure:call:statement*, may be made only from within its scope.

Program names, *item:names*, and *table:names* that are to be defined by *declaration* must be declared before they are used in their respective scopes.

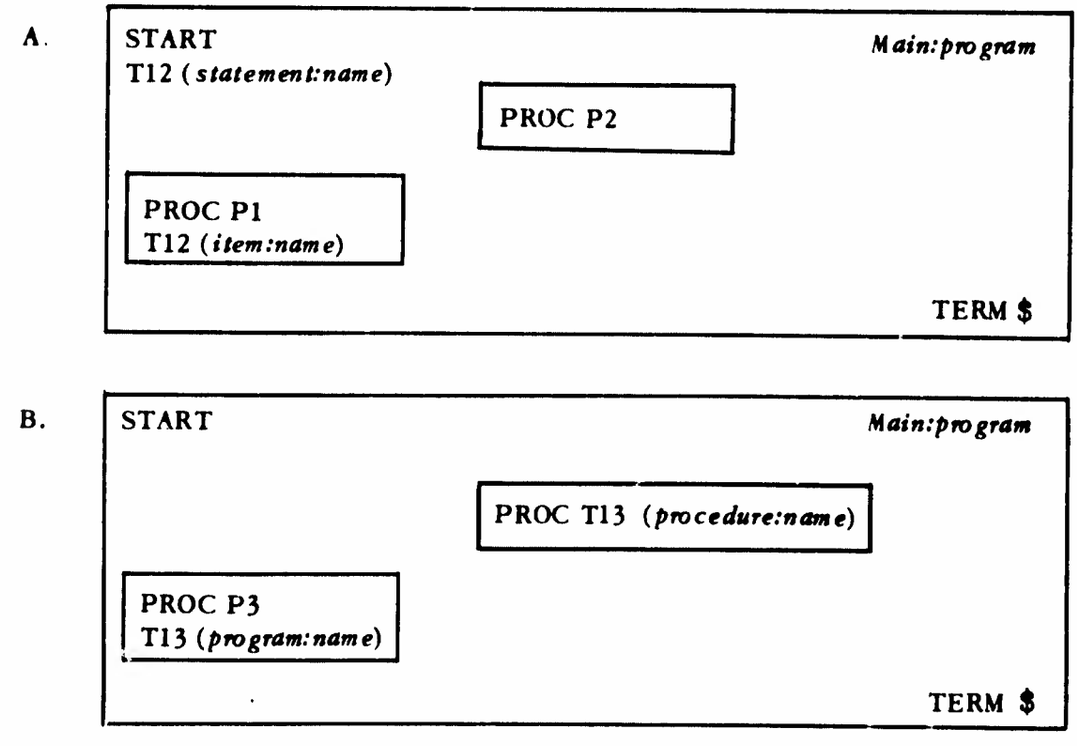


Figure 7. Scope

6.5 LEFT-TO-RIGHT EVALUATION, A SUMMARY

In this and following sections, the order of evaluation is reiterated in detail. The basic rule is that evaluation, combination, and comparison is done from left to right. The basic rule is modified only so far as absolutely necessary for the following circumstances:

1. Differences in precedence of *operators*.
2. Grouping due to *parentheses* and other *brackets*.
3. Dependency on values to the right such as *indices* and *parameters*.
4. An expression known to be irrelevant is not evaluated.

The ordering specified by these rules is for effect only--if there is no difference in the meanings implicit in the *program*, the actual ordering may be different.

6.5.1 LEFT-TO-RIGHT IN *INDICES* AND *PARAMETERS*

An *index* is evaluated before the *item* to which it applies is either set or used. The *index* following BIT or BYTE is evaluated before the *index* following the *item:name* in the case of an *indexed:item*. The components of an *index* are evaluated in strict left-to-right sequence.

In a *function:call* or *procedure:call:statement*, *actual:input:parameters* are evaluated and assigned to the corresponding *formal:input:parameters* from left to right as if specified by a sequence of *assignment:statements*, the *procedure:body* is executed, and the resultant values of the *formal:output:parameters* are assigned to the corresponding *actual:output:parameters* from left to right as if specified by a sequence of *assignment:statements*. The assignment to *actual:output:parameters* is, of course, omitted if the *procedure:body* transfers directly to the *main:program*.

6.5.2 SEQUENCING IN *NUMERIC:FORMULAS*

In evaluating a *numeric:formula*, evaluation of the elements of the *formula* proceeds in strict left-to-right sequence. Specifically, subject to the rules of Section 6.5.1 concerning *indices* and *parameters*, the following elements of a *numeric:formula* are evaluated in the order in which they are encountered while scanning the *formula* from left to right:

1. *constants*
2. *variables*
3. *functions*
4. *numeric:formulas* starting with *primitives*
5. *numeric:formulas* bounded by *parentheses*
6. *numeric:formulas* bounded by the *ideograms* (/ and /)
7. *numeric:formulas* bounded by the *ideograms* (* and *)

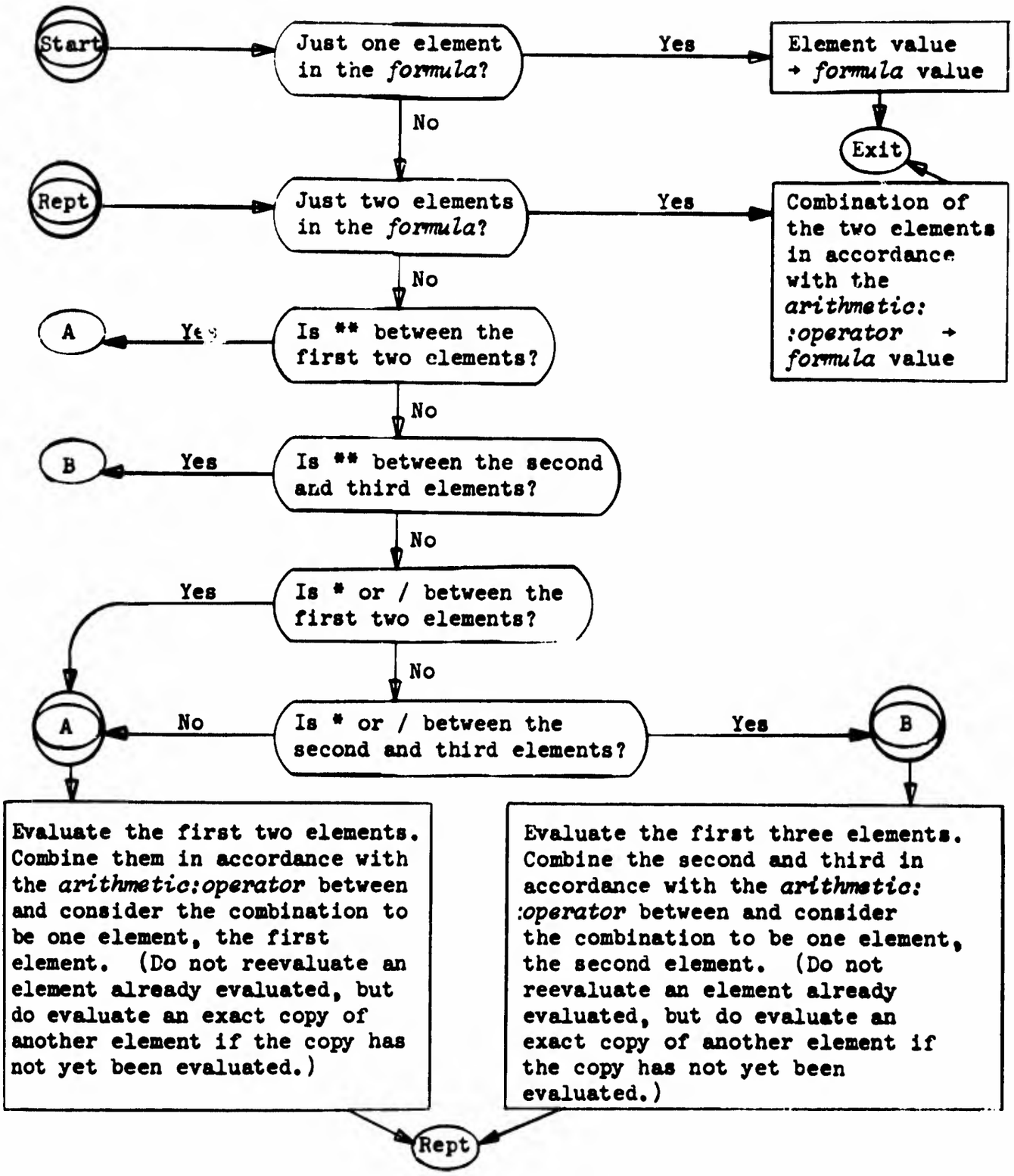


Figure 8. Arithmetic Combination Algorithm

If a *minus:sign* stands immediately before an element such as described above, but not immediately between two such elements it is considered part of the element and reverses the value of the element. A *plus:sign* in such a position is ignored.

For the purposes of further discussion in this section, consider the expression:

8. (* θ *numeric:formula* θ *)

to be replaced by the expression:

9. ** θ (θ *numeric:formula* θ)

Then any *numeric:formula* bounded by *primitives* or *ideograms* other than *arithmetic:operators* consists of a sequence of elements separated by *arithmetic:operators*. The elements of such a *formula* are combined in accordance with the diagram of Figure 8.

6.5.3 SEQUENCING IN *BOOLEAN:FORMULAS*

Any *boolean:formula* bounded by *ideograms* or *primitives* other than AND and OR may be considered to consist of a sequence of elements (*boolean:formulas*) separated by AND's and OR's. It is important, here, to distinguish between *boolean:formulas* and mere fragments. A *numeric:formula* is <u>not</u> one of the elements we are considering here. Neither is an expression with unbalanced *parentheses*.

NOT is always considered part of the element following. Any *boolean:formula* enclosed in *parentheses* is an element. A single element or a sequence of elements separated by AND's but not bounded by AND's may be considered a group. Elements and groups are indicated in Figure 9.

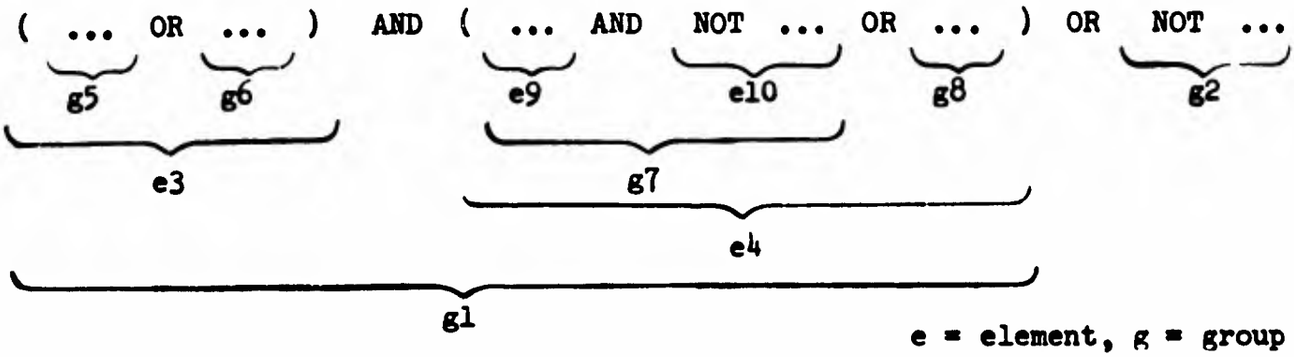


Figure 9. *Boolean* Elements and Groups

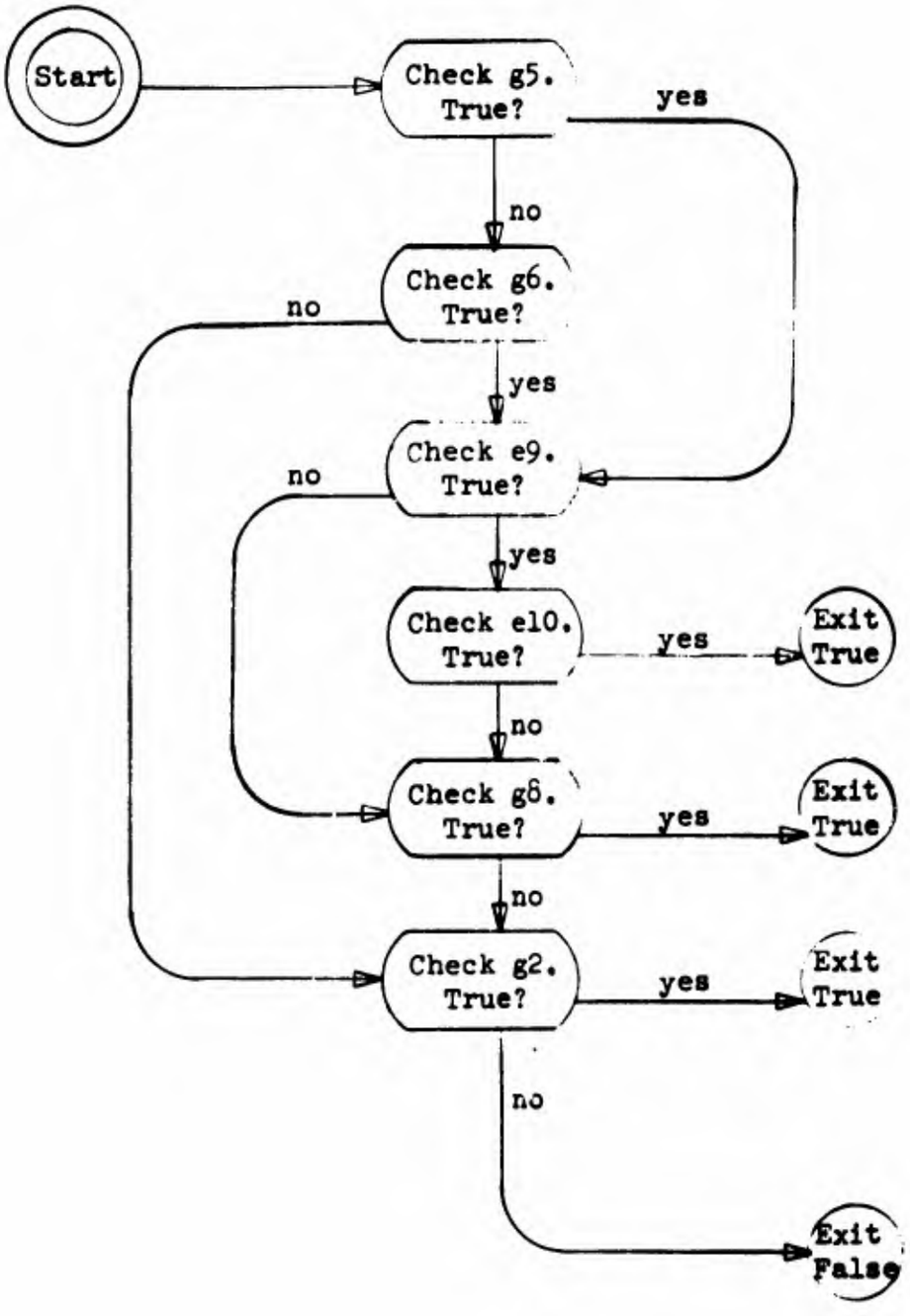


Figure 10. An Example of *Boolean:formula* Evaluation

Evaluation of *boolean:formulas* is subject to the rules of Sections 6.5.1 and 6.5.2. Any *boolean:formula* is a sequence of groups or a sequence of elements. If it is a sequence of groups, groups are evaluated from left to right until one is found to be true, in which case the *formula* is true. If every group is found to be false, the *formula* is false. If the *formula* is a sequence of elements separated by AND's, elements are evaluated from left to right until one is found to be false, in which case the *formula* is false. If every such element is true, the *formula* is true. Evaluation of elements or groups of a *formula* stops when the truth value of the *formula* is determined. The order of evaluation of the example of Figure 9 is illustrated by the diagram of Figure 10.

INDEX AND GLOSSARY

Three kinds of words are indexed below: English or programmer's jargon, metalanguage words and phrases, and JOVIAL *primitives* and *abbreviations*. The metalanguage is in italics and the JOVIAL is in elite all capitals. References are to the numbers of sections where the terms are defined or most fully explained. Defining references marked with a dagger pertain, not to this document, but to TM-555/002, which describes the complete J3 language.

Many terms are defined partially or completely in this index (glossary). Such definitions are intended as reminders for people who are already familiar with the language. For metalanguage terms, consult the defining section to avoid overlooking important exceptions and qualifying remarks. Defining expressions or remarks are indented under the word or phrase to which they apply. Expressions with the same level of indentation are alternate definitions or different ways of saying the same thing. In some cases there are second, or even third, levels of indentation to define the definitions. Lines of the index are numbered at the left except that a line that is merely a continuation of the previous line is not numbered.

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

System Development Corporation
Santa Monica, California

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE

Grammar and Lexicon for Basic JOVIAL

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

5. AUTHOR(S) *(Last name, first name, initial)*

Perstein, M. H.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
10 May 1966	101	

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
U. S. Government b. PROJECT NO.	TM-555/005/00
c. d.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

This is the reference manual for the basic JOVIAL version of the JOVIAL language. Basic JOVIAL is a subset of J3, which is the full language. This manual defines the full syntax and semantics of Basic JOVIAL without drawing fine distinctions between the two aspects.

DD FORM 1 JAN 64 1473

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
JOVIAL						
J3						
Grammar						
Lexicon						
Syntax						
Semantics						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2*a*. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2*b*. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7*a*. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7*b*. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8*a*. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8*b*, 8*c*, & 8*d*. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9*a*. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9*b*. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through

_____."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through

_____."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through

_____."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS)*, *(S)*, *(C)*, or *(U)*

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.