TM-555/002/02A

The JOVIAL Manual, Part 2

The JOVIAL Grammar and Lexicon

16 March 1964

TM-555/002/02A
24 July 1964
combined with
TM-555/002/02

TECHNICAL MEMORANDUM

(TM Series)

This document was produced in connection with a research project sponsored by SDC's independent research program.

The JOVIAL Manual, Part 2

The JOVIAL Grammar and Lexicon

by

Millard H. Perstein

16 March 1964

(Previous version by Christopher J. Shaw)

SYSTEM DEVELOPMENT CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA

MODIFICATION TO:

TM-555/002/02, "JOVIAL Grammar and Lexicon," dated 16 March 1964.

APPROVED

Millard H. Perstein

M. H. Perstein

System Development Corporation/2500 Colorado Ave./Santa Monica, California

Modified Pages		Notes and Filing Instructions
2A	108	The page numbers listed to the left of this column are all new pages, dated 24 July and are to replace the old pages dated 16 March 1964.
3	109	
25	110	
29	113	
42	114	
54	117	
55	118	
57	119	
69	120	
79	121	
81	122	
83	123	
91	124	
93	125	
95	126	
97	127	
100	128	
103	129	
104	130	
105	131	
106	132	
107	133	
	134	

PREFACE

Part 1 of the JOVIAL Manual is an introduction for non-programmers entitled Computers, Programming Languages and JOVIAL. It was written 20 December 1960 by C. J. Shaw and is designated TM-555, Part 1. This document, Part 2, the JOVIAL Grammar and Lexicon, is a complete, concise, and rigorous description of JOVIAL (J3), an SDC-designed, procedure-oriented programming language. It is intended primarily as a specification of the language and is not considered a training document. Three SDC publications on JOVIAL (J3) may be considered textbooks on the complete language:

TM-555/003/00 by C. J. Shaw 26 December 1961
The JOVIAL Manual, Part 3, The JOVIAL Primer, 216 pages.

N-18652/000/00 by Sandra Peterson 24 July 1962
JOVIAL Syllabus, 144 pages.
This is an internal SDC document and is not appropriate for release outside the corporation.

TM-780/000/00 by Phyllis R. Kennedy 17 September 1962
A Simplified Approach to JOVIAL (A Training Document), 387 pages.

There is also an introduction to JOVIAL in several parts which will help to carry the student of JOVIAL a considerable part of the way to an understanding of the complete language:

TM-555/061/00 by M. H. Perstein 8 October 1962
JOVIAL for the Dilettante, Part 1, 40 pages.

TM-555/062/00 by M. H. Perstein 5 November 1962
JOVIAL for the Dilettante, Part 2, 22 pages.

TM-555/063/00 by M. H. Perstein 2 January 1963
JOVIAL for the Dilettante and Beyond,
SDC Compiler Error Detection Lists, 19 pages.

When ordering any of these publications, the user should request all pertinent modifications. TM-555/063/00, the Error Lists, is a handy booklet for all active J3 programmers using SDC compilers. Since the above publications have appeared, new error messages have been added and changes and clarifications to the language have been approved by cognizant committees and implemented by compiler maintenance programmers. This manual includes all changes approved to date. As new changes are implemented, modifications to this document will be issued.

This version of Part 2 is published in the hope of rendering the specification of the language more easily understood. It differs from the previous version principally in the following ways: a different metalanguage, omission of formal division into numbered forms, inclusion of a detailed index, fewer examples, changes of emphasis, and expanded references to implementation. The references to implementation are included mainly to point out the meanings and uses of various elements of the language. These references also serve to remind the reader that it is necessary to consult supplementary documents concerning implementation by the compiler in which he is interested.

The author gratefully acknowledges the help of the following people in critically reviewing a draft of this document: S. L. Arnold, C. Baum, E. R. Clark, V. L. Cohen, E. Hayes, J. S. Hopkins, C. W. Jackson, Jr., P. R. Kennedy, D. K. Oppenheim, and C. J. Shaw. Responsibility, however, for the format, contents, and any errors which may come to light rests entirely with the author.

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

Chapter 1. Introduction

1.1 Language and Metalanguage

In attempting to define and describe programming languages it has been found convenient, perhaps necessary, to use some other language which is richer than the programming language, even richer than the programming language and English combined. Such a richer language is known as a metalanguage.

This document describes JOVIAL in terms of a specially devised metalanguage, but one which is a simplification rather than an elaboration of others that have been used for describing programming languages. The author is interested in knowing if he has accurately gauged the needs of his readers. A short, one-page questionnair is included at the back of the book. Feedback, both negative and positive, will be appreciated.

1.2 The Descriptive Metalanguage for JOVIAL

The language of this document consists of JOVIAL symbols plus English plus other words and phrases plus numbers plus punctuation plus arrangement on the page plus diagrams. Certain elements of JOVIAL look just like the punctuation used with English, for instance the comma and the period which 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" will be distinguished from both JOVIAL words and English words by being typed in a special font called "script." An example, to be defined later, is "*letter*." All such "other words" will be spelled like English words and will have similar, but not identical, meanings. For instance, "*letter*" refers to an element of the JOVIAL alphabet while "letter" refers to an element of the English alphabet.

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

1. In sentences, it is English and part of the sentence.
2. In formulas, it is JOVIAL and part of the expression.

Punctuation may appear indiscriminantly in elite or script, with no meaning attached to the difference.

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

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

3. *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 will never be broken within a word. Here are four examples of phrases (to be defined later) naming classes of JOVIAL structures:

4. *formal:input:parameter:list*
5. *formal:output:parameter:list*
6. *actual:input:parameter:list*
7. *actual:output:parameter:list*

There will be script 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 will be 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 4, 5, 6, and 7, taken two at a time:

8. *input:parameter:list*
9. *output:parameter:list*
10. *formal:parameter:list*
11. *actual:parameter:list*

1.3 Programming Forms and Formats

This document is not concerned with how a *program* gets into the computer. The coding form may be scanned by an optical reader or the manuscript may be transcribed to punched cards or tape. There may be columns reserved for identifying or numbering the cards. The programmer will probably have adopted some consistent and easy-to-read format. This manual, however, considers a JOVIAL *program* to be, from start to termination, 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 JOVIAL *programs*. In those instances when structure or meaning is described as compiler-dependent, the user must consult other documentation (or write it if he is building the compiler) to learn of further restrictions. Since information about JOVIAL compilers is available, this manual also tells about some deficiencies or pathologies in the compilers.

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.

All that this manual requires of a compiler is that it properly compile a legitimate *program*. A good compiler, however, will exhibit the following additional characteristics:

1. It will not stop prematurely nor go wild no matter how indigestible the supposed *program*.

2. It will give clues as to why the supposed *program* is <u>not</u> a *program*.

Such clues are usually called error messages. A good compiler can also be helpful by providing listings of information it has collected and organized concerning the *program*.

Compilers will often not reject certain illegal or undefined structures, but compile them instead, giving results which 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.

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* which comprise the JOVIAL alphabet.

The general order in which the elements of a *program* have been 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 have led 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 be for reference use, the opposite arrangement would be much more difficult for a reader.

An index has been included which will, hopefully, facilitate reference. The index should answer many questions directly. It will carry one quickly back through the chain of definitions until the question is answered or until the reader needs more details, to which he will be directed through the section numbers.

2.2 Spaces and *Spaces*

There is no means in this manual, other than context, of distinguishing between a *space*, an element of JOVIAL, and a space, an element of English and of our descriptive metalanguage. Rather than using a special character for one or the other, it was felt best to make explicit explanations where necessary. The first such explanation follows immediately.

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 will be pointed out in sections 2.5 (*comment*) and 2.63 (*hollerith:* and *transmission:code:* *:constants*). In general, *symbols* are separated by *spaces*. Again the exceptions will be noted (section 2.7), but, note here, 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 2.61).

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 sections 2.7 and 2.8 the transition is noted and forms are explained that don't quite fit the new rule or the old one.

In Chapter 3 and beyond, the opposite view is maintained with respect to *spaces*. From there to the end of the book (except for the index) *spaces* must come between all elements except where declared otherwise.

In the index, neither rule holds. This is a question of detail which the index cannot answer directly.

2.3 *Signs*, Elements of the JOVIAL Alphabet

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

Letter means one of the twenty-six 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 twelve 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*.

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 contructing *programs*. These two categories can be regrouped in ways that are more suggestive of their roles in the language.

Those *symbols* which 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 JOVIAL language:

O ABS	O ENTRY	+ LS	✱ PROC
O ALL	+ EQ	O MANT	✱ 'PROGRAM
+ AND	✱ FILE	O MODE	✱ RETURN
✱ ARRAY	✱ FOR	O NENT	✱ SHUT
✱ ASSIGN	✱ GOTO	+ NOT	✱ START
✱ BEGIN	+ GQ	+ NQ	✱ STOP
O BIT	+ GR	O NWDSEN	✱ STRING
O BYTE	✱ IF	O ODD	✱ SWITCH
O CHAR	✱ IFEITH	✱ OPEN	✱ TABLE
✱ CLOSE	✱ INPUT	+ OR	✱ TERM
✱ DEFINE	✱ ITEM	✱ ORIF	✱ TEST
✱ DIRECT	✱ JOVIAL	✱ OUTPUT	
✱ END	O 'LOC	✱ OVERLAY	
O ENT	+ LQ	O POS	

✱ = STATEMENT ID
O = DATA QUALIFICATION
+ = OPERATOR

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* which 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*. For the *primitives* in the above list spelled without the *prime* an alternate form will be accepted in which a *prime* precedes the *letters*. For example, the following two *symbols* are *primitives* with the same meaning:

1. GOTO
2. 'GOTO

The following *symbol*, however, is not a *primitive*; it may be used as a *name*:

3. LOC

A *name* is a string of two or more *letters*, *numerals*, and *primes* with the following characteristics:

4. It is not identical to any *primitive*.
5. It begins with (the leftmost *sign* is) a *letter*.
6. The rightmost *sign* is not a *prime*
7. It does not contain two consecutive *primes*.

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* which may be used as *abbreviations* will not be exhibited here, but will be 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 20 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 combinations:

8. ($
9. $)

2.6 *Constant*

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

2.61 Adjectives Applying to JOVIAL

Transmission:code means having values which are strings of *signs*, each *sign*, if within a computer, being represented by a string of six bits (binary digits) in accordance with the table of figure 1. In figure 1, octal digits are used to represent patterns of three bits in accordance with the usual convention.

Hollerith means having values which are strings of *signs*, each *sign*, if within a computer, being represented in a manner dependent on the particular computer. In all present versions of JOVIAL, the internal *hollerith* representation uses six bits per *sign*.

Integer, as a noun, means a numeric value which is represented as a whole number without a fractional part, but which 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 any intervening *spaces*.

Fixed means having numeric values, within the computer, with a specific given or stated or understood degree of precision. If the precision is negative it means that the value is stated not even to the nearest unit. *Fixed* does not mean immutable; hence we are led to such seeming incongruities as *fixed:variable*.

Sign	Code	*Sign*	Code	*Sign*	Code	*Sign*	Code
Space	ØØ	K	2Ø	)	4Ø	Ø	6Ø
#	Ø1	L	21	-	41	1	61
#	Ø2	M	22	+	42	2	62
#	Ø3	N	23	#	43	3	63
#	Ø4	O	24	=	44	4	64
#	Ø5	P	25	#	45	5	65
A	Ø6	Q	26	#	46	6	66
B	Ø7	R	27	$	47	7	67
C	1Ø	S	3Ø	*	5Ø	8	7Ø
D	11	T	31	(	51	9	71
E	12	U	32	#	52	'	72
F	13	V	33	#	53	#	73
G	14	W	34	#	54	/	74
H	15	X	35	#	55	.	75
I	16	Y	36	,	56	#	76
J	17	Z	37	#	57	#	77

means there is no corresponding *sign*.

Figure 1. *Transmission:code*

Floating means having numeric values represented within the computer by two numbers. These two numbers are the signicand, which carries 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. A *floating* value is equal to the signicand, multiplied by 2 raised to the power of the exrad. The number of bits in the signicand depends only on the particular computer involved. 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 which are in the nature of *constants*.

Dual means having pairs of numeric values. Each member of the pair is known as a component. The two components must be represented in the same way, each being *octal* in the sense of *integer*, or each being *fixed* with the same precision, or each being *integer*.

Boolean means having one of two possible values which might be thought of as "true and false," or "yes and no," etc., and which are represented by 1 and Ø respectively.

Status means having values which 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 *transmission:code* or *hollerith* or *octal*.

Numeric means *integer* or *fixed* or *floating* or *octal*. In some other discussions of JOVIAL, *numeric* is defined to include *dual*, but, in the hope of making later explanations clearer, *numeric* here excludes *dual*.

Having defined the above adjectives, it will now be 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*, *boolean:variable*, etc., are clear.

2.62 *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.

2.63 The Structure of *Constants*

Integer:constant means a *number*, or a *number* followed by the *letter* E followed by a *scale*. (The E stands for exrad.) An *integer:constant* is a JOVIAL *symbol*. It has a numeric value given, if there is no *scale* present, by reading it as a mathematical symbol. If a *scale* is present, the value of the *integer:constant* is the value of the *number* multiplied by 10 raised to the power given by the *scale*. For example, the following two *integer:constants* have the same value:

1. 2E3
2. 2ØØØ

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

3. *number* .
4. *number* . *number*
5. . *number*
6. *number* . E *optionally:signed:scale*
7. *number* . *number* E *optionally:signed:scale*
8. . *number* E *optionally:signed:scale*

Examples of *floating:constants:*

7. 3.14159
8. 56789.E-3

Fixed:constant means a *floating:constant* followed by the *letter* A followed by an *optionally:signed:scale.* It is a *symbol.* Its value is the value of the *floating:constant* part, curtailed perhaps because of the *optionally:signed:scale* following the A. This *optionally::signed:scale* tells how many bits are to be retained after the point in a binary representation of the value. If the number of bits to be retained is negative, the meaning is that some of the least significant bits to the left of the binary point are to be truncated. On the following three lines are six *fixed:constants.* Although the precision to be carried may be different, the <u>values</u> of the two *fixed:constants* on each line are identical, being that given, in binary, by the third number on the line:

9.	2.A4	2.24AØ	1Ø
10.	4.ØA-2	.6E1A-2	1ØØ
11.	2.25A2	2.375A2	1Ø.Ø1

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:*

12. O(2Ø2Ø2)
13. O(1234567Ø)

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.*

Dual:constant means one of the three structures in the following list:

14. D(*optionally:signed:integer:constant, optionally:signed::integer:constant*)
15. D(*optionally:signed:fixed:constant, optionally:signed::fixed:constant*)
16. D(*octal:constant, octal:constant*)

In the form above in which each component is a *fixed:constant*, the *scale* after the A must be the same in each component. Examples of *dual:constants*:

17. D(27,-15)
18. D(+1.739A1∅,-1.∅92A1∅)
19. D(O(7777),O(4∅76))

Hollerith:constant means a *number* followed by the *letter* H followed by a *left:parenthesis* followed by a string of *signs* followed by a *right:parenthesis*. The value of the *number* must correspond to the number of *signs* between the *parentheses*. The value of a *hollerith:constant* is the string of *signs*, represented within the computer in *hollerith*. The string of *signs* between the *parentheses* may include *spaces*. Examples:

20. 28H(THIS IS A HOLLERITH CONSTANT)
21. 17H(SO IS THIS...+-)$)

Transmission:code:constant means the same as *hollerith:constant* except that the H is replaced by T and the computer representation is in *transmission:code* instead of *hollerith*. Example:

22. 29T(THIS ONE IS TRANSMISSION CODE)

Boolean:constant means the *numeral* ∅, which stands for "false," or the *numeral* 1, which stands for "true." *Boolean:constants* are distinguished from *integer:constants* of the same form by context.

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

23. V(A)
24. V(POOR)
25. V(ALL'GONE)

The value of a *status:constant* depends entirely on context. In each context the *status:constant* will be associated with a *status:item:name* or with a *file:name*. The *status:constants* associated with each *status:item:name* or *file:name* must differ among themselves, but they need not be different from those associated with other *status:item:names* or *file: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*.

2.7 Transition

All the *symbols* of the JOVIAL language have now been explained, at least so far as their structure is concerned. Some meanings have also been explained, but others will be 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 (sections 3.4 and 3.55, for example), but not when used as a *decimal:point* in writing *constants* (section 2.63). Similar remarks concerning + and - might be made, but no ambiguity results from disregarding such commentary. Examples:

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

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

```
6.        BEGIN.5  .6  1.3  2.  END
7.        IF  ALPHA  EQ  7.OR.3Ø2  LQ  BETA  LQ.9ØØ7$
```

In the metalanguage formulas to follow, a *space* will appear between *symbols* wherever a *space* is permitted or required. In examples, *spaces* might not be 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*.

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 *name* '' string of *signs* '' $

Among the *signs* between the first '' and the second '' shown above, there must not be another two *primes* in succession; and the last *sign* before the second set of two *primes* must not be a *prime*. *Spaces*, however, are permitted among the *signs* of the string. In fact, the string may consist of nothing more than a single *space*. There must not be a *comment* between the *name* and the first '' *symbol*.

The *define:directive* is meaningful only if the quoted string of *signs* is actually a string of *spaces* or else a string of *symbols*. Its purpose is to permit a *name* to be used instead of the quoted string of *symbols* at subsequent points in the *program*. Wherever such a "defined" *name* is used it will be effectively replaced by the quoted string of *symbols* with the following exceptions:

2. As part of a *status:constant*.
3. As part of a *literal:constant*.
4. As part of a *comment*.
5. Within *direct:code* other than within *direct:assigns*.

A *name* may be redefined by the use of another *define:directive* for the same *name* at a 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*.

A defined *name* may be included among the *symbols* defining another *name*, effecting the implied replacement. Beyond the second such *define:directive* the effect is the same regardless of the order in which the *directives* were written.

The programmer must avoid circular definitions.

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

Examples of *define:directives*:

```
6.        DEFINE  TO  ''...''  $
7.        DEFINE  GOOD  ''V(GOOD)''$
8.        DEFINE  WORD  ''  ''  $
9.        DEFINE  UNIT  ''D(.7Ø7A8,.7Ø7A8)''$
```

Chapter 3. *Statements*

3.1 Introduction

A JOVIAL *program* consists of a string of *statements* and *declarations* which 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 may itself be named. The terms *string* and *array* are used to characterize certain groups of *items*. For input and output purposes the basic elements are known as *records*, which are grouped into *files*.

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*. There is one kind of element whose value can be changed, but not by means of an *assignment::statement*. This is the *file:name* which, in certain contexts, may be considered to be the *name* of an *item* which contains information about the condition of the *file*. It is not considered a *variable*. Among the JOVIAL *primitives* are some, known as *functional:modifiers*, which 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 the case of *numeric:* or *dual: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* will be discussed in Chapter 4. They can describe these *named:variables* in terms of the adjectives defined previously and listed below:

1. *transmission:code*
2. *hollerith*
3. *integer*
4. *fixed*
5. *floating*
6. *dual*
7. *boolean*
8. *status*

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

3.21 *Simple:variables*

Simple:variable means the *name* of an *item* declared by an *item:declaration* not associated with any *array:declaration* or any *table:declaration.* The adjectives which characterize the *variable* depend on the type description in the *declaration.* Exceptions will be explained later, wherein no *declaration* is required. Example of a *simple:variable:*

1. ALPHA

3.22 *Indexed:variables*

It now becomes necessary to introduce the notion of recursive definitions. We will define *indexed:variable* in terms including *index. Index* will be defined in terms of *formulas* which will be defined later in terms of *variables,* including *indexed:variables.* This is not to be interpreted as 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 *numeric:formulas* separated by *commas.* Each *formula* in the string is known as a component.

Indexed:variable means a structure of the following form:

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

where *name* is the *name* of an appropriately declared *item.*

If the *index* in the above structure has one component, it serves to specify a particular value from a one-dimensional *table* or *array* of values. To pick a value from a two- or more-dimensional *array* requires a two- or more-component *index*. Each time an *index* is evaluated, each component must yield a positive value or zero. If the value is not an integer it will be truncated to an integer. Each component must also, of course, be small enough to specify an actual *entry* of the *table* or *array*. Example of an *indexed:variable:*

1. PAWN ($ RANK, FYLE $)

3.23 Additional *Variables*

As descriptions of *variables*, the terms *floating*, *dual* and *status* may only describe *named:variables*, that is, *simple:* or *indexed: :variables*. The rest, terms such as *hollerith*, *integer*, and *boolean*, however, may be applied to other *variables* which will be explained in the following sections.

3.24 *Integer:variables*

Following is a list of the structures which, along with *named:integer: :variables*, are also *integer:variables:*

1.	*loop:variable*		
2.	BIT	($ *index* $)	(*named:variable*)
3.	CHAR	(*floating:variable*)	
4.	POS	(*file:name*)	
5.	NENT	(*name*)	

The four *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 compiler.

The *functional:modifier*, POS, operating on a *file:name*, designates the position of the *file*. The value Ø corresponds to a position before the first *record* of the *file*, the position form which the first *record* may be read or written. For a *file* of k *records*, the value k corresponds to the position after the last *record*. The value of this *variable* changes when the *file* is involved in input or output. If the value is changed, as by an *assignment:statement*, the *file* is repositioned accordingly. Values outside the range from Ø to k are, of course, undefined.

The *functional:modifier,* CHAR, operating on a *floating:variable,* designates the exrad of the *variable,* a negative, positive, or zero integer value. The term CHAR comes from the common practice of using "characteristic" in lieu of "exrad" by analogy between logarithms and floating numbers.

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.* 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* and with the size of numeric values that the compiler is prepared to handle. The second component may have the value Ø in which case the value of the *variable* is Ø. If only one bit is wanted a one-component *index* may be used, indicating which one.

3.25 *Fixed:variables*

The following structure, as well as the *named:fixed:variable,* is also a *fixed:variable:*

1. MANT (*floating:variable*)

The *functional:modifier,* MANT, operating on a *floating:variable,* designates the signicand of the *variable,* a signed, fixed, fractional value. The term MANT comes from the common practice of using "mantissa" in lieu of "signicand" by analogy between logarithms and floating numbers. Example:

2. MANT (ALPHA ($ 3,5 $))

3.26 *Literal:variables*

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

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

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 6 bits representing a single *sign*. The bytes of an n-byte *literal:item* are indexed from left to right from Ø through n-1. The one- or two-component *index* subscripting 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*. The value of a byte-string of zero length is blanks. The BYTE *variable* is *hollerith* or *transmission:code* if the *named:* *:variable* is *hollerith* or *transmission:code*, respectively. Example:

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

3.27 *Boolean:variables*

Besides the *named:boolean:variable*, the following two structures are also *boolean:variables*:

1. ODD (*loop:variable*)
2. ODD (*named:numeric:variable*)

The *functional:modifier* ODD designates the value true when the least significant bit of the modified *variable* represents a magnitude of one, and false when it represents a magnitude of zero. ODD is true, therefore, when the absolute value of the modified *variable*, considered an integer regardless of the actual type or scaling, is odd; and false when that magnitude is even.

3.28 *Entry:variables*

A *table*, which will be discussed at greater length later, 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 comprise a single *table*. An *entry:variable* is an agglomeration of the values of the *items* comprising an *entry* of a *table*. Its value depends on both the structure of the *entry* and the values of the *items* comprising 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* which can denote the value. Following are the two equivalent forms of the *entry:variable*:

1. ENTRY (*name* ($ *index* $))
2. ENT (*name* ($ *index* $))

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 which characterize *variables* may also be applied to *formulas*. An important kind of *formula* is the *function*.

3.31 *Functions*

A *function* means one of the following structures:

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

A *function* is also known as a *function:call*. *Actual:input:parameter:list* is explained in section 3.55 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 which is *hollerith*, *floating*, *boolean*, etc., and which derives from the computations specified in the *function:declaration* which defines it. Examples:

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

3.32 *Literal:formulas, Status:formulas, Entry:formulas*

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

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

A *transmission:code:formula* means one of the following expressions:

5. *octal:constant*
6. *transmission:code:constant*
7. *transmission:code:variable*
8. *transmission:code:function*

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

9. *status:constant*
10. *status:variable*
11. *status:function*

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

12. Ø
13. *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*.

3.33 *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

Recall that *numeric* means *integer*, *fixed*, *floating*, or *octal*. A *numeric:formula* means one of the following structures:

6. *numeric:constant*
7. *numeric:variable*
8. *numeric:function*
9. NWDSEN (*name*)
10. NENT (*name*)
11. 'LOC (*name*)
12. 'LOC (*name* .)
13. + *numeric:formula*
14. - *numeric:formula*

15. (*numeric:formula*)
16. (/ *numeric:formula* /)
17. ABS (*numeric:formula*)
18. *numeric:formula arithmetic:operator numeric:formula*
19. *numeric:formula* (* *numeric:formula* *)

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 may be 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*. 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. Within groups the order of operations is negations first, then exponentiations, then multiplications and divisions, then additions and subtractions. Within these categories operations are performed from left to right.

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

Conversions between *floating* and *integer* or *fixed* will be carried out automatically as required to perform the indicated operations. The exact order of such conversions and the scaling of intermediate results is dependent on the compiler, which will take into account the characteristics of the target computer and, possibly, the use of the *formula* within the *program*. Division by zero is undefined, as is an exponentiation such as (-2)**.5 which would result in a complex root being taken.

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 non-negative integer equal to the machine address of the core location containing
the *simple:item* or
the first word of the *simple:item*
  or of the *table*
  or of the *named:statement*
  or of the *program*
  or of the first compiler-assigned occurrence of
    the *table:item* or of
    the *string:item* or of
    the *array:item*.

The specified location is not that of any associated control register which may precede the *item*, *table*, *array*, or *program*.

Examples of *numeric:formulas*:

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

3.34 *Dual:formulas*

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

1. *numeric:formula*
2. *dual:constant*
3. *dual:variable*
4. *dual:function*
5. + *dual:formula*
6. - *dual:formula*
7. (*dual:formula*)
8. (/ *dual:formula* /)
9. ABS (*dual:formula*)
10. *dual:formula* *arithmetic:operator* *dual:formula*
11. *dual:formula* (* *dual:formula* *)

Arithmetic:operators, ABS, *parentheses*, and other *brackets* have the same meanings with respect to *dual:formulas* as they have with respect to *numeric:formulas*. A set of computations with *dual* values is carried out as the indicated set of computations with all the left components performed in parallel with the same set of computations on the right components. When necessary a *numeric* value is "twinned" to convert it into a *dual* value so that it can partake in computations with other *dual* values. There is no inverse conversion method for turning *dual* values into *numeric* values. As with *numeric* values, division by zero and the taking of complex roots are undefined.

Examples of *dual:formulas:*

12. THETA ($ ZENDA $) + D(1.ØØ17A15,-1.ØØ63A15)
13. D(-1,1) + COMPLEX'PRODUCT (IOTA, D(Ø.A5,13.719A5))

3.35 *Relational* Operations

A *relational:operator* is the means of expressing a relation between two *formulas.* The relation is in the form of a proposition which 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. In the case of *dual:formulas,* for the proposition to be considered true the relation must hold for both component pairs. If necessary a *numeric* value will be "twinned" for the comparison. The precision of an arithmetic comparison is compiler dependent, but will usually match or better the precision of the least precise of the two values involved.

Between *status:formulas, literal:formulas,* or *entry:formulas,* the truth of the relation depends on the numeric encoding of the values as unsigned integers. If *entry* values are of different lengths, the shorter is prefaced with zeros for the comparison. If *literal* values are of different lengths, a shorter *octal* value is prefaced with zeros, but a shorter *hollerith* or *transmission:code* value is prefaced with properly encoded *blanks.*

A *boolean:formula* is <u>not</u> to be interpreted in terms of its numeric encoding. Hence the *relational:operators* cannot be used to express relations between *boolean:formulas*. Indeed, *relational:operators* are <u>elements</u> of *boolean:formulas*.

A *numeric:relation:list* means one of the structures in the following list:

7.	*relational:operator*	*numeric:formula*
8.	*numeric:relation:list*	*numeric:relation:list*

A *dual:relation:list* means one of the following structures:

9.	*relational:operator*	*dual:formula*
10.	*dual:relation:list*	*dual:relation:list*

A *literal:relation:list* means one of the following structures:

11.	*relational:operator*	*literal:formula*
12.	*literal:relation:list*	*literal:relation:list*

Examples of *relation:lists*:

13. EQ (XYZ + J / 3) / 2
14. LQ ALPHA LQ BETA EQ GAMMA NQ 27.5

<u>3.36 *Boolean:formulas*</u>

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

1.	*boolean:constant*		
2.	*boolean:variable*		
3.	*boolean:function*		
4.	*numeric:formula*	*numeric:relation:list*	
5.	*dual:formula*	*dual:relation:list*	
6.	*literal:formula*	*literal:relation:list*	
7.	*file:name*	*relational:operator*	*status:formula*
8.	*status:variable*	*relational:operator*	*status:formula*
9.	*entry:variable*	EQ	*entry:formula*
10.	*entry:variable*	NQ	*entry:formula*
11.	(*boolean:formula*)		
12.	NOT *boolean:formula*		
13.	*boolean:formula*	AND	*boolean:formula*
14.	*boolean:formula*	OR	*boolean:formula*

In the forms above in which *relation:lists* occur, for the *boolean: :formula* to express the value "true," it is necessary and sufficient that the relation expressed by each *relational:operator* and its two adjacent *numeric:*, *dual:*, or *literal:formulas* be true.

The three *primitives*, AND, OR, and NOT, are known as *boolean:* or *logical:operators*. Their meanings are illustrated in figure 2. In the heading of the figure, p and q stand for simple *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.

From the preceding discussion of *formulas* it can be seen that a *boolean: :formula* may contain *arithmetic:operators*, *relational:operators*, and *logical:operators*. It can be deduced from the previous explanations, but it is well to point out here, that in such a *boolean:formula* the *arithmetic:operators* are applied first, then the *relational:operators*, and finally the *logical:operators*. The obvious exception to this rule is that a *function* must be evaluated before a *formula* in which it is embedded can be evaluated; consequently *relational:operators* among the *parameters* of the *function* will be utilized before *arithmetic:operators* external to the *function*.

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

Figure 2. Effect of the *Logical:operators*

Examples of *boolean:formulas:*

15. 1.5 LQ XX LQ 3.79
16. ALPHA EQ BETA AND NOT LIT LS O(Ø377)
17. (A+B LS C-D OR X+Y GR 2) AND (X EQ Y OR A LS C)

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* will be 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* . *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, they will be 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. *exchange:statement*
3. *go:to:statement*
4. *test:statement*
5. *return:statement*
6. *stop:statement*
7. *procedure:call:statement*
8. *input:statement*
9. *output:statement*
10. *open:input:statement*
11. *open:output:statement*
12. *shut:input:statement*
13. *shut:output:statement*

3.51 *Assignment:statements*

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

1.	*numeric:variable*	=	*numeric:formula*	$
2.	*dual:variable*	=	*dual:formula*	$
3.	*literal:variable*	=	*literal:formula*	$
4.	*boolean:variable*	=	*boolean:formula*	$
5.	*status:variable*	=	*status:formula*	$
6.	*entry:variable*	=	*entry:formula*	$

Assignment:statements can be further characterized, in the obvious way, by means of the adjectives which occur in each of the above six 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 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 throughout the calculations needed to evaluate the *formula*. A *function* may, of course, be included in the *formula*. Evaluation of a *function* may involve side effects, which possibility will become apparent when we consider *function::declarations*. If the side effects of evaluating the *function* involve other elements of the *formula* in which the *function* is embedded, the results are undefined. This is so because, although the rules for evaluation of a *formula* are unequivocal concerning the order in which elements are combined, the order in which they are mobilized is not stated, except that each *statement* is completely executed before anything is done about the next *statement*. In evaluation of *numeric::formulas* and *dual:formulas*, rules have already been stated concerning conversions to compatible forms among *integer*, *fixed*, *floating*, and *dual* values. Such conversions will also be carried out where necessary in assigning the value as required in *numeric:* and *dual:assignment::statements*.

In executing *literal:assignment:statements* there will not be any conversion among *hollerith*, *transmission:code*, and *octal* values. If the value of the *formula* is longer than the *literal:variable* to which it is to be assigned, excess bytes will be truncated starting at the left end. If shorter, *blanks* will be added at the left, as required, coded in *hollerith* or *transmission:code* to match the coding of the *literal::variable*.

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* (or *mode:directive*) which previously described 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* which differs in length from the *entry:variable* on the left, in making the assignment excess machine registers are truncated starting at the left or else full registers of zeros are added at the left to make up the deficiency.

3.52 *Exchange:statements*

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

1.	*numeric:variable*	==	*numeric:variable*	$
2.	*dual:variable*	==	*dual:variable*	$
3.	*literal:variable*	==	*literal:variable*	$
4.	*boolean:variable*	==	*boolean:variable*	$
5.	*status:variable*	==	*status:variable*	$
6.	*entry:variable*	==	*entry:variable*	$

Exchange:statements can be further characterized by means of the adjectives which occur in the above expressions. For instance, *dual:exchange:statement* means the expression on the second line of the above list.

The *exchange:statement* specifies that the old value of each of the two *variables* is to become the new value of the other *variable*. The remarks made in connection with *assignment:statements* concerning conversion of *numeric* values, non-conversion of *literal* values, and truncation and augmentation of *literal* values and *entry* values apply also to *exchange:statements*, but in both directions. Example:

```
7.        ENT (T1($I$)) == ENT(T1($Ø$))$
```

3.53 *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*	($	*index*	$)

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 *sequence:designator* $

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 will not occur if the *sequence:designator* does not lead, perhaps circuitously, to a *statement:name*, a *program:name*, or the *name* of a *close:declaration*, and the next *statement* executed will therefore be 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 invoked it. Finally, if the *sequence:designator* is, or leads eventually to, a *statement:name*, the interruption of the *statement* execution sequence will be permanent, with the next *statement* executed being the one bearing the specified *statement:name*.

3.54 *Test:statement, Return:statement, Stop:statement*

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

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

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 3.77).

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

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 compiler environment includes an operating system, the *stop:statement* may be compiled so as to return control to the operating system. Or it may be that only the *stop:statement* without reference to a *statement:* *:name* will return control to the system. If the computer halts without giving control to the system and if it is then restarted by some means, the execution sequence will resume with the next *statement* listed, or with the *statement* bearing the specified *statement:name* if one is given in the *stop:statement*. See section 6.1 for the use of *stop:statements* in "other" *programs*.

3.55 *Procedure:call:statements*

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

1. *formula*
2. *array:name*
3. *table:name*
4. *close:name* .
5. *actual:input:parameter:list* , *actual:input:parameter:list*

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* which would provide a value meaning for the *status:constant*.

Note that a *close:name* in an *input:parameter:list* is identified as such by the presence of a following *period*.

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

6. *variable*
7. *table:name*
8. *array:name*
9. *statement:name* .
10. *actual:output:parameter:list* , *actual:output:parameter:list*

Note that a *statement:name* in an *output:parameter:list* is identified as such by the presence of a following *period*.

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

11. *procedure:name* $
12. *procedure:name* ( ) $
13. *procedure:name* (*actual:input:parameter:list*) $
14. *procedure:name* (= *actual:output:parameter:list*) $
15. *procedure:name* (*actual:input:parameter:list* = *actual:output:*
    :parameter:list) $

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* which bears the same *name*. That is *table:name*, *close:name*, *statement:name*, and *formula*, *variable*, or *item:name* must correspond to *table:name*, *close:name*, *statement:name*, and *item:name*, respectively. In the *procedure:declaration* the *names* listed as *formal:parameters* are referenced elsewhere in the *declaration*. The execution of the *procedure* is effected as if all such references to *table:* *:names*, *array:names*, *close:names*, and *statement:names* were replaced by the corresponding *actual:parameter:names*. This extends to the *items* of *tables* which are named as *formal:parameters*. That is, references to the variously named bits of the *formal:parameter:table* will be effected as references to the corresponding bits of the *actual:parameter:table*. The above description is of the intended method for handling these *parameters*, but in at least one version of the compiler complete sets of values are transferred between the *tables* which are named as *actual:parameters* and *formal:parameters*.

With respect to *parameters* which are *formulas* and *variables*, execution of the *procedure* is effected as if the values of the *formulas* which are *actual:input:parameters* are assigned to the *items* which are *formal:input::parameters* before execution and the values of the *formal:output:parameters* are assigned to the *variables* which 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 3.51). *Indices* in the *actual:parameter:lists* are evaluated before execution.

3.56 Input, Output, and *Files*

With many data storage devices the insertion or withdrawal of the value of an arbitrary item of information may be a relatively complex operation, requiring the transfer of an entire block or *record* of data. Such devices are termed "external" storage devices, as contrasted with the "internal" memory of the computer. To allow a reasonably efficient description of algorithms involving the data stored in an external storage device, the *file* concept is introduced, so that all data which enter or leave the internal memory of the computer are organized into *files*.

A *file* is a collection of *records* each of which is again a collection -- of bits or bytes depending on the *file* type: binary or Hollerith. A *file* of length k may be considered a vector, arranged as follows:

$$p(\emptyset),\ R_{\emptyset},\ p(1),\ R_1,\ \ldots\ ,\ p(k-1),\ R_{k-1},\ p(k)$$

where the R's are *records*, the components of the vector, and the p's are partition symbols, with a computer-dependent physical representation, which may be interpreted as:

$$p(k) = \text{end-of-file};\quad p(n < k) = \text{end-of-record}.$$

If the *record* currently available for transfer to or from the *file* is R_n, the *file* is positioned at partition symbol p(n), and the value designated by "POS(*file:name*)" is n. An *assignment:statement* "POS(*file:name*) = N $" positions the *file* to the value specified by N, where $\emptyset \le N \le k$. In particular, "POS(*file:name*) = Ø $" "rewinds" the *file*. Any *file* for which the general positioning operation is to be avoided as inefficient (e.g., tape) or impossible (e.g., cards, printer) is called a "serial," as opposed to "addressable," *file*.

A *record* in a *file* may be input by a read operation or output by a write operation, although some *files* are read-only or write-only depending on the characteristics of the storage device involved.

Input:operand means one of the following five expressions:

1. *variable*
2. *array:name*
3. *table:name*
4. *table:name* ($ *index* $)
5. *table:name* ($ *index* ... *index* $)

Output:operand means one of the following two expressions:

6. *constant*
7. *input:operand*

An *operand* in an *input:statement* or an *output:statement* specifies the *record* to be read or written, which may consist of the bits or bytes representing: a single value, denoted by a *constant* or a *variable;* the values comprising an *array*, indicated by an *array:name;* the values comprising a *table*, indicated by a *table:name;* the values comprising a *table:entry*, indicated by a *table:name* subscripted by a 1-component *entry:index;* the values comprising a consecutive set of *table:entries*, indicated by a *table:name* subscripted by a pair of one-component *entry:indices* (separated by ... the continuation *ideogram*), whose values specify the initial and final *entries* of the set.

3.57 *Input:statements, Open* and *Shut*

Input:statement means:

1. INPUT *file:name* *input:operand* $

Open:input:statement means one of the two following expressions:

2. OPEN INPUT *file:name* $
3. OPEN INPUT *file:name* *input:operand* $

Shut:input:statement means one of the two following expressions:

4. SHUT INPUT *file:name* $
5. SHUT INPUT *file:name* *input:operand* $

A *file* may be read by the execution of a sequence of *statements* consisting of, first, an *open:input:statement*, next, a sequence of *input:statements*, and finally, a *shut:input:statement*.

An *open:input:statement* activates the *file* and prepares it for reading. An *open:input:statement* need not designate that a *record* be read, in which case, *file* position is initialized to zero. If, however, values are designated to be read from a *record*, the read operation is initiated and *file* position is set to 1. The meaning of "initialized to zero" depends on the compiler and the characteristics of the *file*. It may mean "set to the initial position" or it may mean "call the present position zero."

An *input:statement* initiates a read operation transferring data from the *record* to represent the designated values, and increments the *file* position by 1. The sequence of *statement* executions may continue, concurrently, with the read operation although the *file* is "busy" (or at any rate not "ready") until the read is successfully terminated. This occurs when a partition symbol is encountered, or when all the designated values have been read from the *record*. A read operation is unsuccessful when started from the end-of-file position or when uncorrectable errors occur in the data transmission.

A *shut:input:statement* serves to deactivate the *file*, releasing the external storage device associated with the *file* for possible other use. A *shut:input:statement* need not designate values to be read from a *record*, but if any are designated, the read operation is completed prior to the deactivation of the *file*.

3.58 *Output:statements*, *Open* and *Shut*

Output:statement means:

1. OUTPUT *file:name* *output:operand* $

Open:output:statement means one of the following two expressions:

2. OPEN OUTPUT *file:name* $
3. OPEN OUTPUT *file:name* *output:operand* $

Shut:output:statement means one of the following two expressions:

4. SHUT OUTPUT *file:name* $
5. SHUT OUTPUT *file:name* *output:operand* $

A *file* may be written by the execution of a sequence of *statements* consisting of, first, an *open:output:statement*, next, a sequence of *output:statements*, and finally, a *shut:output:statement*.

An *open:output:statement* activates the *file* and prepares it for writing (e.g., an identification block may be written). An *open:output:statement* need not specify that a *record* be written, in which case *file* position is initialized to zero. If, however, an *output:operand* is specified, the write operation is initiated and *file* position is set to 1.

An *output:statement* initiates a write operation for the next output *record* and increments the *file* position by 1. The sequence of *statement* executions may continue, concurrently, with the write operation, although the *file* is not "ready" again until the write is successfully terminated, when all the specified bits or bytes are written without the occurrence of any uncorrectable error in the data transmission. In some *files*, partition symbols and thus *file* positions are predetermined. Consequently, a write operation started from the end-of-file position would be unsuccessful. In other *files*, notably tape *files*, the partition symbols are determined by the write operation itself so that, in effect, the end-of-file partition symbol follows the last *record* written.

A *shut:output:statement* serves to deactivate the *file*, causing its termination by an end-of-file partition symbol and releasing the external storage device associated with the *file* for possible other use. A *shut:output:statement* need not specify that a *record* be written, but if an *output:operand* is specified, the write operation is completed prior to the deactivation of the *file*.

The *records* of a *file* have no internal structure, and may be thought of as strings of bits or bytes. Structure is supplied only by the *operand* portion of the *input:statement* or *output:statement*. Thus, reading and writing are just information transfers, and no editing or rearranging of data (except that required for conversion to 6-bit *hollerith* code) is implied. A write transfers just the bits or bytes specified by the *operand*. A read transfers just the bits or bytes of the *record*, to the maximum designated by the *operand*.

A *shut:statement* is defined only for active *files*, and an *open:statement* is defined only for inactive *files*. Further, some *file* pairs must not be active concurrently, for example: two *files* on the same tape reel.

Input:statements and *output:statements* are defined only for active *files*, and in general, an active *file* may be both written and read, and positioned -- if the *file* characteristics allow. Thus, a read with a serial, write-only *file* such as a printer is undefined. The characteristics of some *files*, however, also preclude the initiation of a read or write operation when the *file* is "busy", thus eliminating the possibility of stacking input-output operations.

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* *statement:list*
3. *directive* *statement:list*
4. *statement:list* *declaration*
5. *statement:list* *directive*
6. *statement:list* *statement:list*

A *compound:statement* means the following expression:

7. BEGIN *statement:list* END

Example of a *compound:statement:*

```
8.        BEGIN ALPHA = 1 $ SL1. GOTO SL7 $ SL2. BEGIN INT (
          XØ, X1, DERIV. = AREA) $ GOTO DERIV $ END END
```

3.7 *Complex:statements*

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

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

3.71 *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 which is processing the *program*. For a precise definition of *direct:statement* it will be 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 A(*optional:optionally:signed:integer:constant*) = *named:variable* $
2. ASSIGN *named:variable* = A(*optional:optionally:signed:integer:constant*) $

There must be no *spaces* between the A and the *left:parenthesis* or between the *parentheses*. In the first form above, the value of the *named:variable* is moved to the accumulator (the principal program-accessible register of the arithmetic unit). In the second form the value is moved from the accumulator to the *variable*. If there is no *constant* within the *parentheses* the contents of the accumulator represent a *floating* value. If there is a *constant* other than zero the value is *fixed*, with the stated number of fractional bits in the accumulator. A negative number means the binary point is so many places to the right of the right end of the accumulator. If the *constant* is zero the accumulator contains an *integer* or non-*numeric* value.

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:

3. *signs*
4. *direct:assign*
5. *direct:code* *direct:code*

Direct:statement means:

6. DIRECT *direct:code* JOVIAL

Although *direct:code* is arbitrary so far as the definition of JOVIAL expressions is concerned, only certain configurations will be meaningful. If the input medium is punched cards, specifications of meaningful *direct:code* will probably involve positioning on the card. Because of this it will probably be "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.

It has been felt "safest" to classify a *direct:statement* as a *complex::statement*, but if it contains no *direct:assigns* it may be considered a *simple:statement*.

3.72 *Conditional:statements*

A *conditional:statement* means:

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

Remember that an *independent:statement* is a *simple:statement* or a *compound:statement*. The expressions in this and following sections which are here called *clauses* are known as *statements* in other JOVIAL documentation. The present nomenclature, however, is felt to make it easier to understand the language structure.

If:clause means one of the two following expressions:

2. IF *boolean:formula* $

3. *statement:name* . *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.

Following are two examples of *conditional:statements:*

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

3.73 *Alternative:statements*

Whereas a *conditional:statement* provides an *independent:statement* which may or may not be executed depending on the satisfaction of a condition, an *alternative:statement* provides a list of *independent::statements* and associated conditions. That *independent:statement* associated with the first condition which is satisfied will be the only one executed if any one is. The conditions are expressed by the *boolean:formulas* in the following definitions.

If:either:clause means:

1. IFEITH *boolean:formula* $

Or:if:clause means:

2. ORIF *boolean:formula* $

Alternative means one of the expressions in the following list:

3. *or:if:clause* *independent:statement*
4. *statement:name* . *alternative*

Alternative:list means one of the following two expressions:

5. *if:either:clause* *independent:statement* *alternative*
6. *alternative:list* *alternative*

Alternative:statement means:

7. *alternative:list* END

Here is one example of an *alternative:statement:*

8.
```
IFEITH       ALPHA  LS  BETA  $
             ALPHA  =   BETA  $
L1 . ORIF    ALPHA + BETA  GR  1Ø  $
     BEGIN      GAMMA = ( ALPHA + BETA ) / 2  $
             L2 .  ALPHA = GAMMA + 1  $
                   BETA  = GAMMA + 1  $
     END
   ORIF  1  $  GOTO  KEEP  $
END
```

The above example provides for the execution of one *assignment:statement* if the first condition is satisfied. It makes no difference then if any of the other conditions are satisfied: after execution of the single *assignment:statement* the execution sequence continues with the *statement* following the second END. If the first condition is not satisfied, the second condition is examined, and so forth. The third condition in the example is a catch-all. The *constant* 1 is a *boolean:formula* which always has the value "true." A jump to L1 from elsewhere in the *program* will cause the search for *alternatives* to begin at that point; it is as if execution of the *alternative:statement* had begun at the top, but that all the conditions before the referenced *name* were false. A jump to L2 will cause execution of the *statement* at that point regardless of the satisfaction of the earlier conditions. In this case only two of the three *simple:statements* which comprise the *independent:statement* of this *alternative* will be executed. Following execution of BETA = GAMMA - 1 $ control will pass to the *statement* following the *alternative:statement*.

Although Ø and 1 should be recognized as *boolean:formulas* in *if:clauses*, *if:either:clauses*, and *or:if:clauses*, the compilers presently recognize only 1 in the expression ORIF 1 $ in such cases. Actually, of course, this is the only place, other than *assignment:statements*, where such recognition is useful.

3.74 *Loop:statements*

The *loop:statement* provides for the "iteration" of an *independent:statement* (or *special:compound*). 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*. Remember that a *loop:variable* is a single *letter* in certain contexts. Those contexts will now be described.

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

1. FOR *loop:variable* = *numeric:formula* , *numeric:formula* , *numeric:formula* $
2. FOR *loop:variable* = ALL (*name*) $

In the second of the above expressions the *name* must be a *table:name* or the *name* of an *item* belonging to a *table*. In either case, the *complete:for:clause* with the ALL *modifier* is equivalent to either one of the following two expressions:

3. FOR *loop:variable* = Ø , 1 , NENT (*name*) - 1 $
4. FOR *loop:variable* = NENT (*name*) - 1 , - 1 , Ø $

The designers of each compiler are free to decide, arbitrarily, which of the two interpretations to select. Presumably they will choose that interpretation which is likely to give the better machine language code. Hence, the ALL *modifier* should be used only when the programmer does not care which of these two interpretations is assumed.

The *complete:for:clause* defines a *loop:variable* to control the iteration of an *independent:statement* and for use 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 reached or 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 whether the increment value is currently positive or negative, respectively. In some compilers the direction of comparison depends on the explicit sign rather than the current value of the increment.

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

5. FOR *loop:variable* = *numeric:formula* , *numeric:formula* $

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 will have to be provided by some other means.

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

6. FOR *loop:variable* = *numeric:formula* $

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

Special:compound means one of the following expressions:

7. BEGIN *statement:list* *if:clause* END
8. *name* . *special:compound*

Although the *special:compound* is not, strictly speaking, a *statement*, the *name* in the second of the above two expressions is a *statement::name*. It may be considered a *name* of the first *statement* in the *statement::list*. The *special:compound* may take the place of the *independent::statement* in a *loop:statement* and be iterated under control of the *loop:variables*.

Incomplete:loop:statement means one of the following expressions:

9. *incomplete:for:clause* *independent:statement*
10. *incomplete:for:clause* *special:compound*
11. *incomplete:for:clause* *incomplete:loop: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:

```
12.        FOR  I = 1 , I  $
     SL1.  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:loop:statement means one of the following expressions:

13. *complete:for:clause* *independent:statement*
14. *complete:for:clause* *special:compound*
15. *complete:for:clause* *incomplete:loop:statement*
16. *one:factor:for:clause* *complete:loop:statement*

From the last two definitions we see that a *loop:statement* is a string of *for:clauses* followed by an *independent:statement* or a *special:compound*. A *special:compound* may be used as part of a *loop:statement* only if at least one of the string of *for:clauses* is a *two:factor:for:clause* or a *complete::for:clause*. In a *complete:loop:statement* it is actually permissible for more than one of the *for:clauses* in the string to be *complete:for::clauses*. The compiler, however, will ignore the third *formula* in all but the first of such *clauses*, treating them as *two:factor:for:clauses*.

3.75 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* or *special:compound* repetitively. Since a *loop:statement* is a *statement*, it may be part of the *statement::list* which 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* which forms its latter part.

Each *for:clause* defines or activates the *loop:variable* which 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* which 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 which activated it. It is even active and may be used in the one or two *formulas* following the *formula* which provides its initial value in the same *for:clause* that activates it. A *for:clause* may be used to activate only a *loop:variable* which 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 will be 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* on any *for:clauses* other than the first one in a string as well as to *statement:names*, *switch::names*, or *close:names* on or within the *independent:statement* forming the latter part of the *loop:statement*. It is permitted to transfer control to *statements*, *for:clauses*, *switch:declarations*, and *close::declarations* within a *loop:statement* from other points within the same *loop:statement*.

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 the bottom 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* or through one of the *actual:parameter* alternate exits (from a *procedure*) if this alternate exit is a *name* within the *loop:statement*.

3.76 *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.

Loop:variables are, on the other hand, defined within *switch:declarations* and *close:declarations* 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*. If a *procedure:call:statement* or a *function:call* within a *loop:statement* contains, as an *input:parameter*, the *name* of a *close:declaration* also within the *loop:statement*, this is considered proper invocation of the *close:name* from within the same *loop:statement*.

3.77 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 reached or 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 3.54), *test:statement* means one of the two following expressions:

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

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.

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 which precede it in the initialization sequence for the *loop:statement*. If the referenced *loop:variable* is one which was defined by a *one:factor:for:clause*, control is nevertheless transferred to the proper place so that incrementation and testing takes place for those *loop:variables* defined in the *loop:statement* <u>before</u> the referenced *loop::variable* but not for those defined <u>after</u> the referenced *loop:variable*.

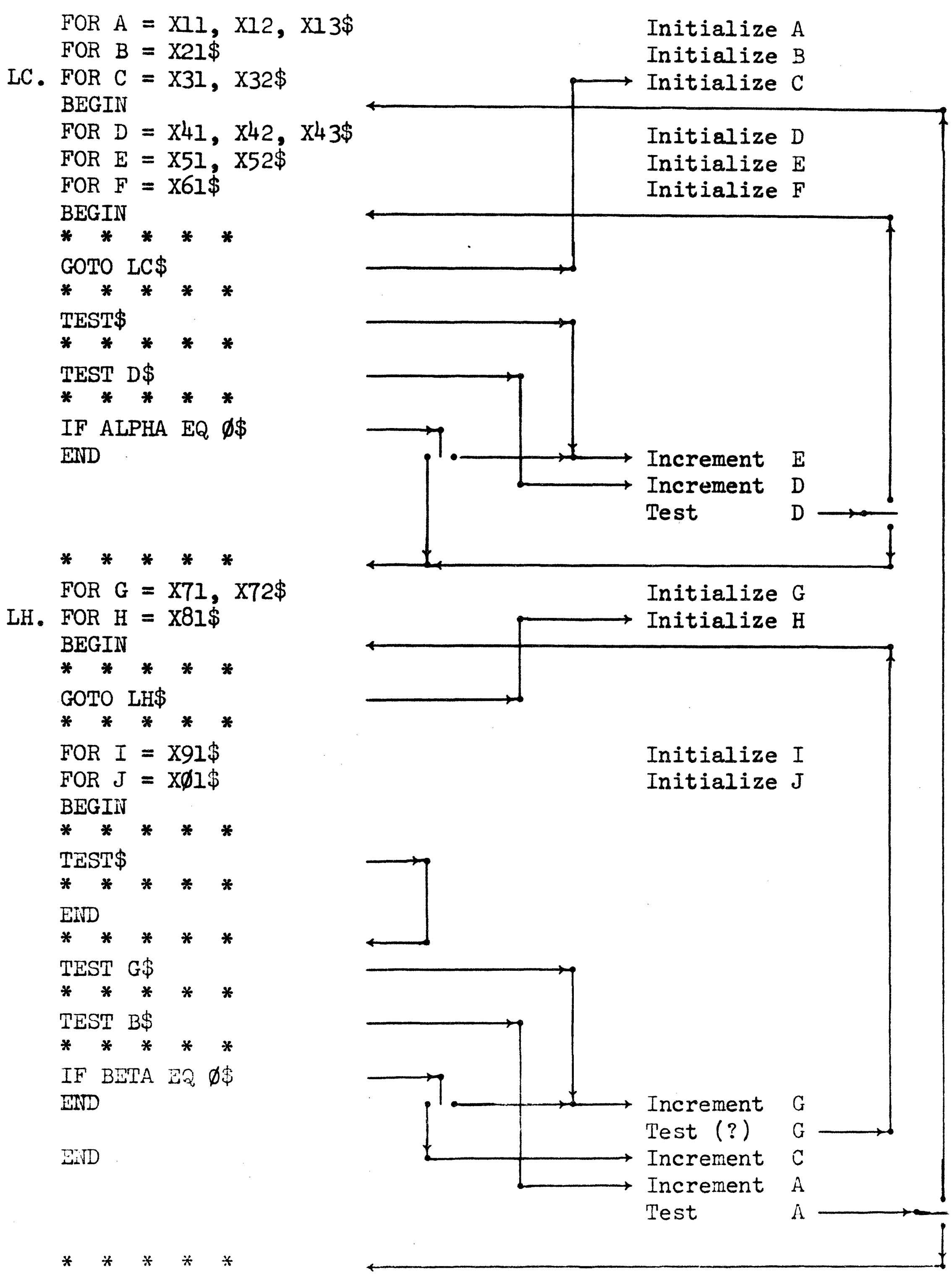

Figure 3. *Loop statement* Summary Example

The *if:clause* at the end of a *special:compound* controls execution of the iteration control routine at the end of the *loop:statement*. Execution of a *test:statement*, however, supersedes control by this *if:clause*. When the *if:clause* is executed , if the condition is true the iteration control routine is executed; if the condition (the *boolean:formula*) is false the iteration control routine is skipped, thus terminating execution of the *loop:statement*.

Figure 3 summarizes the foregoing discussions of *loop:statements*, showing *complete:* and *incomplete:loop:statements*, nested *loop:statements*, *test::statements*, and transfers to the midst of a string of *for:clauses*, together with initialization of the *loop:variables* and iteration control. On the left in the diagram is a sample of JOVIAL coding. A line with five asterisks represents one or more lines of JOVIAL coding not germane to this discussion. On the right the initialization and loop control is shown in positions corresponding to their respective parts of the code. The incrementing and testing of D and E (and F if there were any) corresponds to the first END. The control with respect to A and C (and to B if there were any) corresponds to the last END. The arrows show transfers of control. Some of the coding represented by lines of asterisks must, of course, permit jumps around the *go:to:statements* and *test:statements* shown.

The four little "electric switch" diagrams represent conditional transfers. After the incrementation of G there is an unconditional transfer to the beginning of the loop in which G and H are defined. This transfer point is called a "test on G," but the jump is unconditional since the *loop::statement* is incomplete (there is no limiting formula). In the loop on I and J there is no incrementation and no return since all the pertinent *for:clauses* are *one:factor:for:clauses*.

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 (not in a *parameter:list*) followed by a *period*. It is thereby defined as the *name* of that point in the *program* which is the beginning of the next *statement* or *clause*. A reference, in the *procedure:* or *function:declaration*, to a *name* which is followed by a *period* in a *formal:input:parameter:list* is treated as a reference to the corresponding *close:name* in the invoking *procedure:call:statement* or *function* call. Such reference to a *name* which is followed by a *period* in a *formal:output:parameter:list* is treated as a reference to the corresponding *statement:name* in the invoking *procedure: :call:statement*.

4.2 Pre-defined *Names*

Names may be pre-defined for a *program* as *names* of *items*, *tables*, *files*, external *programs*, *procedure:declarations*, or *function:declarations*. Such pre-definition 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 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.

4.3 Mode-defined *Names*

Names which have not been pre-defined, nor declared, nor previously defined by mode, as elements with conflicting scope and category (section 6.5) may be defined by mode simply by referencing the *name* in an appropriate *statement*. Each compiler assumes a normal mode for such definitions -- probably an integer of some convenient size; perhaps signed, perhaps unsigned. The method for changing this mode is described in section 6.4.

Definition by mode can be done only for *simple:items* and it cannot be done if it would thereby change the existing scope of definition of the same *name* applied to a different element. Consider, for instance, a *procedure:declaration* in which a particular *name* is not declared and is not used as a *formal:parameter*, but is used as if it were the *name* of a *simple:item*:

1. If the *name* has been pre-defined as a *simple:item::name*, or declared in the *main:program* (before this *procedure:declaration* is encountered) as a *simple::item:name*, or defined by mode in the *main:program* (before this *procedure:declaration* is encountered), then the reference to it is a reference to that *simple:item* which is already defined (global *item*).

2. If the *name* has not yet been defined in any way for the *main:program* as any entity in the same category as a *simple:item:name*, then the reference to it in this *procedure:declaration* serves to define it by mode but only within this *procedure:declaration* (local *item*).

3. If the *name* has already been defined in some way for the *main:program*, not as a *simple:item:name*, but in the same category (see section 6.5), then this reference in this *procedure:declaration* is erroneous.

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* which 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 subdivided into groups as follows:

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

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

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

And *indexed:item:declaration* may be subdivided into the following groups:

7. *array:item:declaration*
8. *table:item:declaration*

Numbers, which have been defined in section 2.62, are used extensively in *data:declarations*. In the expressions to be discussed below, there will be 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*:

9. *n1n*
10. *n2n*
11. *n3n*
12. *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* will be used with only one significance in the explanations to follow.

4.41 *Item:descriptions*

Item:descriptions are parts of *item:declarations* which give the characteristics of the *items*. The adjectives, defined in section 2.61, which apply to *constants* and *variables* also apply to *items*, *item::descriptions*, and *item:declarations*.

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

1. F
2. F R
3. F *floating:constant* ... *floating:constant*
4. F R *floating:constant* ... *floating:constant*

The *abbreviation* F specifies a *floating:item*. The *optional:abbreviation* R declares that any value assigned to the *item* be rounded instead of truncated. The pair of *floating:constants* separated by the ... *ideogram*, if present, state an estimated minimum through maximum absolute value range. This range might be used by the compiler in optimizing the machine language program. The *constants* must be positive or zero and the smaller must come first.

Integer:specifier means one of the expressions in the following list:

5. I *n7n* S
6. I *n7n* U
7. A *n7n* S
8. A *n7n* U

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

9. *integer:specifier*
10. *integer:specifier* R
11. *integer:specifier* *integer:constant* ... *integer:constant*
12. *integer:specifier* R *integer:constant* ... *integer:constant*

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*;

R declares, if present, that any value assigned to the *item* be rounded instead of truncated; the optional pair of *constants* give the range as explained for *floating:item:description*. The use of A instead of I in an *integer:item:description* is allowed because of the similarity to a *fixed:item:description*, where A stands for "arithmetic."

Fixed:specifier means one of these two expressions:

13.	A	*n7n*	S	*optionally:signed:n8n*
14.	A	*n7n*	U	*optionally:signed:n8n*

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

15.	*fixed:specifier*				
16.	*fixed:specifier*	R			
17.	*fixed:specifier*		*constant*	...	*constant*
18.	*fixed:specifier*	R	*constant*	...	*constant*

Again, *n7n* declares the number of bits required by the *item*, including the sign bit if there is one; S declares a signed *item;* U declares an unsigned (positive) *item; n8n* declares the number of fractional bits in the *item;* R declares rounding instead of truncating; the pair of *constants* give the absolute value range as explained above. The *constants* may be *floating* or *integer* or *fixed*. The rules about *spaces* permit a form such as 1....5 to be written. This may seem ambiguous, but the necessity for going from lesser to greater values requires it to mean the same as if 1. ... 5 had been written. If the first *number* in such an expression were zero, the meaning would be ambiguous without some convention. Hence the convention is adopted that if *number*.... is written and if nothing <u>preceding</u> forbids, it will be considered as if *number*. ... had been written, whatever may follow the fourth *period*. If *n8n*, along with its optional sign, specifies a negative value, it means that low order integer bits are missing from the *item*. The *abbreviation* A used in this *description* means "arithmetic."

Dual:specifier means one of these two expressions:

19.	D	*n7n*	S	*optional:optionally:signed:n8n*
20.	D	*n7n*	U	*optional:optionally:signed:n8n*

Dual:item:description means one of the following four expressions:

21.	*dual:specifier*				
22.	*dual:specifier*	R			
23.	*dual:specifier*		*dual:constant*	...	*dual:constant*
24.	*dual:specifier*	R	*dual:constant*	...	*dual:constant*

The *abbreviation* D specifies a *dual:item; n7n* declares the number of bits in each component of the *item,* including the sign bits if present; S declares each component to be signed; U declares each component to be unsigned; *n8n* declares the number of fractional bits in each component; R declares rounding instead of truncating; and the optional pair of *dual:constants* declare estimated minimum through maximum absolute value ranges for the two components.

Hollerith:item:description means H *n7n*

and *transmission:code:item:description* means T *n7n*

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

Status:item:description 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 will be 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.*

Boolean:item:description means the *abbreviation* B

4.42 *Simple:items*

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

1. ITEM *name* *item:description* $
2. ITEM *name* *item:description* P *optionally:signed:constant* $
3. ITEM *name* *optionally:signed:constant* $

The *simple:item:declaration* defines an *item* by naming it and describing it. The second and third forms above also give it an initial value, the value of the *constant.* In the second form, the P stands for "pre-set." The *constant* must be consistent with the *item:description;* that is, it must be of a type which can be assigned to this *item* in an *assignment:statement.* In the third form the description is implicitly that of the *constant.* The third form, the one without an explicit *description,* cannot be used to declare *status:items* or *boolean:items;* the use of Ø or 1 or an *octal:constant* declares an *integer:item.*

Examples of *simple:item:declarations:*

1. ITEM ALPHA F $
2. ITEM THETA F R Ø. ... 3.1416 $

```
3.      ITEM  X2  I  6  S  R  5 ... 23 P -18  $
4.      ITEM  X3  A 15  U  5  P  97.168  $
5.      ITEM  X4  -97.168A7  $
6.      ITEM  DX5  D  13  S -3  D(24,24)...D(24ØØØ,24ØØØ)  $
```

4.43 *Independent:overlays*

Space for *items* is allocated in the computer in various ways depending on the particular compiler. In general, space at least as large as the declared size is set aside. There are restrictions, however. Any restriction on the size of a *literal:item* usually is dictated by the maximum size of *n7n* expected by the compiler. Besides this restriction, most compilers will not handle *items*, other than *literal:items*, greater in size than one machine word.

It is possible to specify that storage for *simple:items*, *tables*, and *arrays* 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 machine 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 *literal:item*, for instance, to overlay more than one *item*.

In all compilers (which compile this version of JOVIAL) presently in existence, *simple:items* are not packed. That means they occupy one or more machine words without sharing any with other *items*. Some other features of data storage are not quite so standardized. In one compiler all *tables* begin with a control word containing the number of *entries* in the *table*. In another compiler, only variable length *tables* have this control word. In one compiler *literal:items* have a similar control word; in another compiler they do not. It is often necessary to be cognizant of the presence or absence of control words and of the allocation algorithms used by the compiler when specifying *data:sequences*.

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

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

Array:name is a synonym for *array:item:name*. (*Table:name* is <u>not</u> a synonym for *table:item:name*.)

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

5. *independent:data:sequence*
6. *independent:overlay:specification* = *independent:data:sequence*

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

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

An *independent:overlay:declaration* may be used to arrange *simple:items*, *tables*, and *arrays* in sequence; to overlay these *sequences* on one another; and to assign these overlays to specific machine locations. Within the *overlay:declaration*, data structures separated by *commas* will be given sequential locations in the order in which they are named and *sequences* separated by *equal:signs* will begin at the same location. If the *overlay::declaration* contains a *number* or an *octal:constant* the common origin of the *sequences* will be the location identified by the value of the *constant;* otherwise the common origin will be selected by the compiler not to conflict with other data or program storage. Examples:

10. OVERLAY WORD'LIST = DUMMY, MESSAGE $
11. OVERLAY 1Ø24 = UMPIRE $

The *name* of a data structure may appear no more than once in an *overlay::declaration*, but it may appear in more than one *overlay:declaration* if logical inconsistencies are avoided. With most compilers the avoidance of logical inconsistencies means that any structure named in more than one *overlay:declaration* must immediately follow the *primitive* OVERLAY in all *overlay:declarations* in which it appears, <u>other</u> than the first.

With some compilers, if a data structure derives its location, either directly or indirectly, from an *overlay:declaration* containing a *constant*, it must not be provided with initial values. With all present compilers, data structures named in an *overlay:declaration* must first be defined: either pre-defined by COMPOOL, declared, or defined by mode.

4.5 Complex Data Structures

It is often necessary to specify more complex data structures than *simple::items*. *Tables* and *arrays* serve this need. An *array* is a one-(or more) dimensional arrangement of *items* all having the same *item:name*. The particular *item* out of the *array* is designated by means of an *index* having a number of components corresponding to the dimensionality of the *array*. 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, or it might be element 5 of a one-dimensional *array*. There are exceptions in the structure of a *table:entry*. For instance, a *string:item*, consisting of a linear arrangement of components called *beads*, can only be part of a *table:entry*. Thus a particular *bead* of a particular *string* in a particular *entry* of a *table* would require the *string:item:name* and a two-component *index* for complete identification. (*String* is a synonym for *string:item*).

4.51 *Constant:lists*

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

A *k:plus:one:dimensional:constant:list* is BEGIN followed by a string of *k:dimensional:constant:lists* followed by END. Below are three examples, a *one:dimensional:constant:list*, a *two:dimensional:constant:list*, and a *three:dimensional:constant:list*.

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

```
2.        BEGIN  BEGIN  V(HI'OUTSIDE)  V(HIGH)   V(HI'INSIDE)  END
                 BEGIN  V(OUTSIDE)     V(STRIKE) V(INSIDE)     END
                 BEGIN  V(LO'OUTSIDE)  V(LOW)    V(LO'INSIDE)  END
          END

3.        BEGIN  BEGIN  BEGIN  Ø  1 1 1  Ø  END
                        BEGIN  1  Ø Ø Ø  1  END
                        BEGIN  1  Ø Ø Ø  1  END
                        BEGIN  1  Ø Ø Ø  1  END
                        BEGIN  1  Ø Ø Ø  1  END
                        BEGIN  1  Ø Ø Ø  1  END
                        BEGIN  Ø  1 1 1  Ø  END
                 END
                 BEGIN  BEGIN  Ø  1 1 1  Ø  END
                        BEGIN  1  1 1 1  1  END
                        BEGIN  1  1 Ø 1  1  END
                        BEGIN  1  1 Ø 1  1  END
                        BEGIN  1  1 Ø 1  1  END
                        BEGIN  1  1 1 1  1  END
                        BEGIN  Ø  1 1 1  Ø  END
            END  END
```

At the present time the compilers are prepared to handle *constant:lists* of no more than three dimensions.

4.52 *Arrays*

In specifying an *array* it is necessary to state the number of dimensions and the extent of each dimension. This is done by means of a *dimension::list*. *Dimension:list* means a string of *numbers*. *Array:declaration* means one of the expressions in the following list:

1. ARRAY *name* *dimension:list* *item:description* $
2. ARRAY *name* *dimension:list* *item:description* $ *constant:list*

The *name* is the *array:item:name*. The number of *numbers* in the *dimension::list* is the number of dimensions of the *array*. A one-dimensional *array* is a column vector. (Of course, the programmer may treat it as a row vector if he wishes). A two-dimensional *array* is a matrix, a row of column vectors. A three-dimensional *array* is a set of matrices. And so forth. The (first) *number* in the *dimension:list* declares the number of elements in (each column of) the *array*. The second *number* in the *dimension::list* declares the number of columns in a matrix or plane of the *array* (or the number of elements in a row). The third *number* is the number of planes in a volume. And so forth. The number of dimensions of *arrays* is, of course, limited by what the compiler is prepared to handle. Some compilers do not handle *arrays* at all.

The *item:description* in an *array:declaration* applies to the whole *array*, to each element or component of the *array*. Thus, one might declare an *array* of *boolean:items* or a *dual:array*, where every element of the *array* is a *dual* value with the same number of bits per component and the same number of fractional bits per component.

If the *array:declaration* contains a *constant:list* it must be of the dimensionality declared by the *dimension:list*. However, it need not specify an initial value for every element. The values given are used to set elements starting with the first element of the *array*. Thus, if we wished to specify only the first element of the first column of the front matrix of a 3 by 3 by 3 *array*, the *constant:list* might be as follows:

```
3.        BEGIN   BEGIN   BEGIN   5   END   END   END
```

to specify the middle element of such an *array*, it is necessary to specify other elements leading to it, as follows:

```
4.        BEGIN   BEGIN   BEGIN   1    END   END
                  BEGIN   BEGIN   2    END
                          BEGIN   3 5  END   END   END
```

The 1, 2, and 3 specify initial values for elements we didn't care about, but we had to specify them in order to get a 5 initially into the center of this 27-element *array*. The 1 is the initial value of the upper left hand corner of the front plane (see figure 4). While looking at the figure, the reader should consider the order of indexing into this *array*. The components of the *index* used in referring to an *array:item* are in the same order as the dimension *numbers* in the *dimension:list*. Thus, the *entries* marked A, B, and C in figure 4 are indexed as follows (in JOVIAL, indexing starts with Ø, not 1):

5. A: 2,1,Ø
6. B: Ø,2,Ø
7. C: Ø,1,2

The pre-set values 1, 2, 3, and 5 are indexed as follows:

8. 1: Ø,Ø,Ø
9. 2: Ø,Ø,1
10. 3: 1,Ø,1
11. 5: 1,1,1

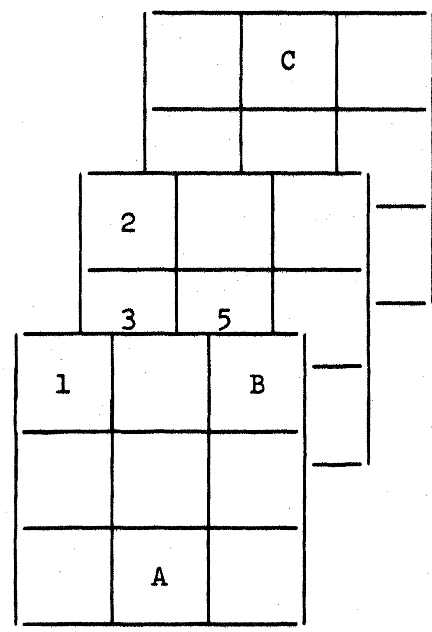

Figure 4.
A 3 by 3 by 3 *Array*

It may appear that the index shown above with the value 3 should be Ø,1,1 instead of 1,Ø,1. That would be true if the bracketing order within a *constant:list* matched the order of components in an *index* and the order of *numbers* in a *dimension:list*. It was felt desirable, however, to match two conflicting conventions. The order of components in an *index* is in accordance with conventional mathematical notation. It is also desirable to write the elements of a *two::dimensional:constant:list* in the same arrangement in which they would appear in a picture of the *array* (compare the arrangement of 2, 3, and 5 in the example with the arrangement in figure 4). In order to do this it was necessary to interchange the bracketing of rows and columns in *constant:lists* of two or more dimensions. Thus, in such a *constant:list* the innermost BEGINs and ENDs bracket elements of rows rather than of columns. BEGINs and ENDs at the second level outward bracket rows of a matrix or plane. BEGINs and ENDs at the third nesting level outward bracket planes of a volume. And so forth. Nothing in the language specifies the order in which the elements of an *array* are to be stored in the computer. This is a compiler-dependent feature of the object code.

4.53 Type Matching for Pre-set 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 which 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 figure 5.

In *fixed:constants* given as initial values of *integer: fixed:* or *floating:items*, the *scale* after the A is ignored, the number of fraction bits to be utilized being picked up from the *item::description*. Therefore, there is really no need to use *fixed::constants* in a *constant:list*.

Example:

```
1.      ARRAY  MTRX   3     4    F     $
        BEGIN  BEGIN  1    -1    Ø     Ø    END
               BEGIN -1.5   2.3  1.7   Ø    END
               BEGIN  Ø     Ø    1.2  -1    END   END
```

Item type	*Constant* types
Integer, fixed, or *floating*	*Integer, fixed, floating,* or any mixtures of these three types
Integer, fixed, or *floating*	*Octal*
Dual	*Dual*
Hollerith or *transmission::code*	*Hollerith, octal,* or *transmission::code*, but not mixed
Status	*Status*
Boolean	*Boolean*

Figure 5. *Constant* Types for Pre-setting *Items*

4.54 *Tables*

The structure of a *table* is more complex than that of an *array* although the possible number of dimensions of a *table* is more limited. In the complex 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*.

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

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

There are subordinate *declarations* which can be used only within *table::declarations*. These subordinate *declarations* will be explained in sections 4.55 and 4.57 prior to putting them together into the three kinds of *table:declarations*.

4.55 *Ordinary:entries*

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

1. ITEM *name* *item:description* $
2. ITEM *name* *item:description* $ *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 4.41 and 4.42). 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.        ALPHA   ($  Ø  $)
4.        BETA    ($  I  +  5  $)
5.        MESSAGE ($  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* ∅, 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 4.56), 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, at the start of execution, of *items* for which no initial values have been provided depend on the particular compiler.

Subordinate:data:sequence means one of the following expressions:

6. *ordinary:table:item:name*
7. *subordinate:data:sequence* , *subordinate:data:sequence*

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

8. *subordinate:data:sequence*
9. *subordinate:overlay:specification* = *subordinate:data:sequence*

Subordinate:overlay:declaration means

10. OVERLAY *subordinate:overlay:specification* $

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

11. *ordinary:table:item:declaration*
12. *ordinary:entry:description* *ordinary:entry:description*
13. *ordinary:entry:description* *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 have been previously declared in *item:declarations* occurring earlier in the same *entry:description*. An *ordinary:entry::description* names and describes all the *items* which comprise a *table:entry*. A *subordinate:overlay:declaration* within the *entry:description* arranges *items* of the *entry* -- *items* separated by *commas* will be allocated consecutive storage space within the *entry* in the order in which they are named, except that packed *items* may be rearranged for storage efficiency; *sequences* separated by *equals:signs* will begin at the same location within the *entry*. Note that a *subordinate:overlay:declaration* cannot specify an absolute location as origin of the *data sequences*.

A *name* may appear in more than one *subordinate:overlay:declaration*, but as with *independent:overlay:declarations*, logical inconsistencies must be avoided. Some compilers do not permit *subordinate:overlay::declarations* since the effect can be achieved by other means (sections 4.57 and 4.58).

4.56 *Ordinary:tables*

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

1. V *n1n*
2. R *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*) is a *numeric:formula* but not a *variable*. It is dependent on the particular compiler whether the value of a variable NENT (*table:name*) is zero, *n1n*, or undefined prior to being set by an *assignment:statement* or some other *statement*.

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 storage cells (computer 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* storage cells, each block containing one word from each *entry* of the *table*. (See figures 6 and 7.) In addition, each *table*, regardless of its basic structure, may contain one or more control words. The presence of control words depends on the particular compiler and whether the *table* is variable or rigid in size.

Serial Structure	Parallel Structure
MN control word	MN control word
1st half AB[Ø]	1st half AB[Ø]
2nd half AB[Ø]	1st half AB[1]
XY[Ø]	1st half AB[2]
1st half AB[1]	1st half AB[3]
2nd half AB[1]	2nd half AB[Ø]
XY[1]	2nd half AB[1]
1st half AB[2]	2nd half AB[2]
2nd half AB[2]	2nd half AB[3]
XY[2]	XY[Ø]
1st half AB[3]	XY[1]
2nd half AB[3]	XY[2]
XY[3]	XY[3]

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

13 consecutive computer words are shown in each illustration above.

Figure 6. Serial & Parallel *Table* Structure

The use of a *basic::structure:specification* in a *table:declaration* is completely optional. In the absence of a *basic:structure:specification*, whether the *table* has parallel or serial structure depends on the compiler. In *tables* with variable length *entries* (section 4.58), the basic structure must be serial, either declared or normal for the particular compiler.

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 usually means that each *item* occupies its own computer word or words without sharing it (or them) with other *items*. "No packing" does not preclude "overlaying." D stands for "dense packing" which may mean that *items* are packed together to utilize every bit in the word and minimize space. The interpretation in many compilers will relax "dense packing" somewhat. For instance, it is likely that an *item* which will fit in one word will not be packed so as to be

Serial Structure

	MN control word			
⋮	2nd word	5th word	8th word	11th word
AB ⋮	3rd word	6th word	9th word	12th word
XY	4th word	7th word	1∅th word	13th word
	Entry ∅	*Entry* 1	*Entry* 2	*Entry* 3

Parallel Structure

	MN control word		
Entry ∅	2nd word	6th word	1∅th 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••••••••••••••		XY

This illustrates the same example as figure 6. The same 13 words are shown, but the blocks are drawn side by side.

Figure 7. Parallel and Serial *Table* Structure

divided between two words and it may be that every *item* which will not fit in one word will begin at the left end of a word. M stands for "medium packing" which usually means that *items* are packed but not so closely that they share sub-register units. Sub-register units are the parts of a computer word which can easily be referenced in machine instructions: -- parts such as half words, addresses, bytes, etc.

Ordinary:table:declaration means the following structure:

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

The *table* need not be named if there is no occasion to refer to it, such as in an *input:statement* or an *independent:overlay:declaration.* 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  64 ... 4Ø95  $
                OVERLAY  ALF = NUM  $
        END
```

4.57 *Defined:entries*

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

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

The elements of the above expression which are also included in the *ordinary:table:item:declaration* (section 4.55) serve the same purpose here that they serve there. This *declaration* occurs only within a *table:declaration* (section 4.58) 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 4-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 have been explained before. The *packing:specification* may be used to provide information which may be useful to the compiler. It does not direct the packing of the *item*, but describes the packing which results from *n3n, n4n*, the *item:description*, and the situation of adjacent *items* in the *entry*.

String:item:declaration means the following expression:

2. STRING *name item:description n3n n4n*
optional:packing:specification n5n n6n $
optional:two:dimensional:constant:list

The *string:item:declaration* provides the means by which an *item* can be declared, having more than one occurrence per *entry* of a *table*. Each such occurrence of a *string:item* is called a *bead*. To refer to a particular *bead* in a *numeric:formula* or a *statement* (*beads* need not be *numeric*) the *item:name* is used, followed by a two-component *index* as in the following example:

3. ALTITUDE ($ K-3 , 5 $)

The first component, K-3 in the above example, indicates which *bead* within the *entry*. The second component, 5 in the example, tells which *entry* of the *table*.

N3n and *n4n*, as in the *defined:entry:item:declaration*, tell in which word in the *entry* and in which bit in the word the *item* (that is, the first *bead* of the *item*) begins. The *optional:packing:specification* describes rather than directs the packing. *N5n* declares the frequency of occurrence of the *string:item* in the words of the *entry*. That is, there are *beads* of the *string* in every *n5n*th word of the *entry* (starting with word *n3n* of course). *N6n* declares the number of *beads* in each word of the *entry*. Consider the following example:

4. STRING ZEUS I 12 U Ø 1 2 3 $

The above example declares that the *beads* of ZEUS are 12-bit unsigned integers, that the first *bead* starts in word Ø, bit 1 of the *entry*, that there are *beads* in every second word of the *entry*, and that there are 3 *beads* in each word of the *entry* which contains *beads*. *N4n* (1 in this example) tells in which bit of the word is the beginning of the first *bead* in each word which contains *beads*. That is, not only does *bead* Ø begin in bit 1 of word Ø, but also *bead* 3 begins in bit 1 of word 2. Suppose that the example is embedded in a *table* declared to have 3 words per *entry* and that on this computer there are 48 bits per word. Then figure 8 may be considered to be a partial illustration of *entry* 4 of the *table*.

A *two:dimensional:constant:list*, if present, declares initial values for some of the *beads* of the *string*. The first *one:dimensional:constant:list* provides values for *beads* in *entry* Ø; the second *one:dimensional:constant:list* provides values for *entry* 1; etc. Within each *one:dimensional:constant:list*, the first *constant* provides the initial value for *bead* Ø; the second *constant* provides the value for *bead* 1; etc.

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

5. *defined:entry:item:declaration*
6. *string:item:declaration*
7. *defined:entry:description* *defined:entry:description*

In other words, a *defined:entry:description* is a string of *string:item: :declarations* and *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.

4.58 *Defined:entry:tables*

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

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

The *table* need not be named if there is no occasion to refer to it. 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*.

Notice that *n2n* is now required, to state the size of an *entry*. Since the *string:item:declaration* declares an unlimited number of *beads*, the size of an *entry* can only be determined by this explicit means. 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 an example, figure 7. Suppose the *program* containing the *table* shown in figure 8 has a reference to ZEUS ($6, 4 $). By the structure of the *table* there is no such *bead*. The compiler, however, will use the *table* structure to determine the beginning of *entry* 4 and the *item* structure to determine the position of *bead* 6 with respect to the beginning of *entry* 4. Hence the reference will be to a *bead* of ZEUS in what would seem to be the middle word of *entry* 5.

Such a reference as we described in the last paragraph is permitted. A *table* to which such references are made is considered to have variable length *entries*. It is even permissible to declare a *string:item* or *table:item* to begin in a word of the *entry* which, by *n2n*, doesn't exist. That is, *n3n* may be equal to or greater than *n2n*. Again, the *table* structure will be used to find the beginning of a referenced *entry* and *n3n* will be used to find the *item*.

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.

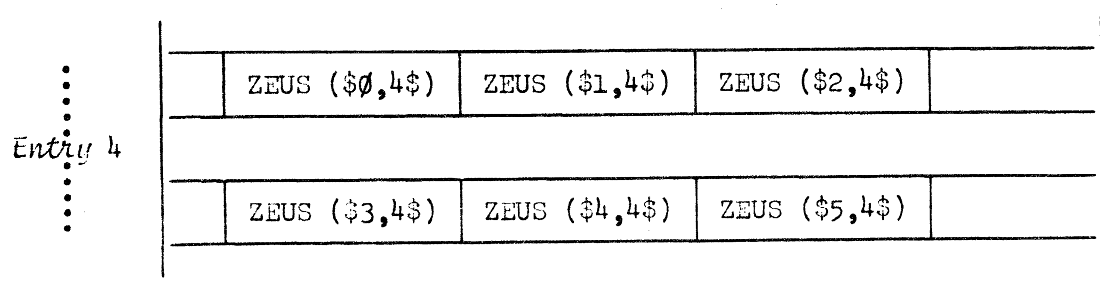

TABLE R 1Ø S 3 $ BEGIN STRING ZEUS I 12 U Ø 1 2 3 $ END

Figure 8. *Entry* 4 of a *Table*

Among the things which must be considered:

2. The *table* must be of serial *entry* structure. This is so even if there is only one word per *entry*. Consideration of figure 7 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*.

3. 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 $) actually occupies that space, but the compiler doesn't know it. When referring to the next ALPHA that does exist, it should be called ALPHA ($ 8 $) and not ALPHA ($ 7 $).

4. 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
```

5. 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*.

4.59 *Like:tables*

Like:table:declaration means the following expression:

1. TABLE *name* *optional:table:size:specification* *optional:basic:structure:specification* *optional:packing:specification* L $

The L just before the *dollar:sign* declares this to be a *table* with an *entry* structure "like" a previously declared and named *table* (or a pre-defined *table*), the "pattern" *table*. The *name* of the *like:table* is formed by suffixing a *numeral* or *letter* to the *name* of the pattern *table*. The *items* of the *like:table* are then automatically named with the *item:names*, similarly suffixed, of the pattern *table*. The composition and structure of the *entries* of the *like:table* are generated by the *declarations* describing the *entries* of the pattern *table*, with the difference, of course, of the *numeral* or *letter* attached to each *item:name*. *Table:size*, *basic::structure*, and *packing* can be specified for the *like:table*, but if this information is omitted, the *specifications* of the pattern *table* are used for these characteristics.

4.6 *Files*

A *file:declaration* is used to name and describe a *file* on some hardware device used for input and/or output and to declare the *status:constants* to be used in designating the various statuses of the *file*. A few preliminary definitions are required.

File:structure:specification means one of the following expressions:

1. H *n9n* V *n1Øn*
2. B *n9n* V *n1Øn*
3. H *n9n* R *n1Øn*
4. B *n9n* R *n1Øn*

Status:list means a string of one or more *status:constants*.

File:declaration means the following expression:

5. FILE *file:name* *file:structure:specification* *status:list* *device:name* $

The *file:name* is a *name* chosen by the programmer and hereby declared as a *file:name*. *Hollerith:file* structure, declared by an H, means that the *records* of the *file* are composed of characters or bytes encoded in whatever manner is normal for the storing of alpha-numeric information on the device. Input or output involving *hollerith:files* and *hollerith:* or *transmission:code:variables* is permissible; any necessary code conversions will be included automatically. If the *variables* are not *hollerith* or *transmission:code*, however, the effect of input/output with a *hollerith:file* is undefined. Input/output with a *binary:file*, declared by a B, and any kind of *variable* is accomplished without any code conversion. The *number* *n9n* declares the estimated maximum number of *records* in the *file*. The *abbreviation* V declares a variable *record* size; the *abbreviation* R declares a rigid *record* size; and *n1Øn* declares the (estimated maximum) number of bits in a *record* (for a *binary:file*) or bytes in a *record* (for a *hollerith:file*).

The various possible statuses of a *file*, such as "busy," "ready," and "error," are associated with numeric values by the compiler. The programmer declares a *list* of *status:constants* to be defined, respectively, as these values, starting at zero and going up by ones. These *status:* *:constants* are then meaningful only in context with the *file:name*, which functions as a *status:item:name*, but only as specified for *file:names* in *boolean:formulas* and in *switch:declarations*. There must be at least one *status:constant* in this *list*, but there need not be as many as there are meaningful values.

The *device:name* must be in the form of a JOVIAL *name*, but the programmer does not choose it. Such *names* are permanently assigned to the various input/output devices available. The use of one of these *names* as a *device:name* does not prejudice its use in some other way, such as an *item:name* or *statement:name*, but it would be incompatible with definition of the *name* by means of a *define:directive*.

One should consult the documentation for a particular compiler for the list of *device:names* and the statuses which apply to the various input/output devices.

Example of a *file:declaration:*

```
6.        FILE    SNAP    H    2ØØ     V   12Ø
                  V(READY)   V(BUSY)  V(ERROR)    V(EOF)
                  TAPE5    $
```

Chapter 5. *Processing:declarations*

5.1 Introduction

Unless otherwise directed (section 6.2) every *program* begins execution with the first *statement* of what may be called the *main:program*. *Statement* execution then proceeds sequentially except for iterations of *loop:statements* and jumps due to *go:to:statements*, *conditional::statements*, and *alternative:statements*. In (almost) every *program*, however, there are groupings of *statements* or other elements of the *program* to which execution control cannot or must not 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* which is blocked from sequential access only because of the presence of *go:to:statements*, *conditional::statements*, or *alternative: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*

Program:declaration means one of the following expressions:

1. 'PROGRAM *name* $
2. 'PROGRAM *name* *number* $
3. 'PROGRAM *name* *octal:constant* $

Notice that the *primitive* introducing the above expressions 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. The *integer* or *octal::constant* declares the machine address of the beginning compiled location of the named *program*. The presence or absence of the *constant* depends on the compiler and the operating system in which it is embedded. If the system supplies the machine location and if it is not desired or not permitted to override this given location, the *constant* 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 will return control to the *statement* following the *go:to:statement;* and that the values of any *loop:variables* which are active at execution of the *go:to:statement* will be 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 *sequence:designators*, but *program:name* is not permitted among them. These *sequence:designators* 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 which are to be executed only when the *switch:name* is invoked by a *go:to::statement* or another *switch*. Therefore, a *switch:declaration* should occur only in a position, relative to *statements*, such that ordinary sequential execution cannot reach it; for example, in the positions indicated below:

3.

```
    .
    .
    .
    STOP  $
    switch:declaration
    .
    .
    .
    XYZ = 5  $
    GOTO statement:name  $
    switch:declaration
    .
    .
    .
```

If, in some system, the *stop:statement* shown above dashes any hope of continuing, then the positions shown for *switch:declarations* are all right. If, on the other hand, it is possible to continue in sequence after the stop, then the first *switch:declaration* in the example should not be in such a position. Some versions of the compiler, however, always prevent "falling into" a *switch*.

A *switch:declaration* within a *loop:statement* must not be invoked by a *go:to:statement* nor via another *switch:declaration* outside that *loop::statement*. For more details see section 3.76 on *processing:declarations* within *loop:statements*.

5.31 *Index:switches*

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

1. *sequence:designator*
2. , *sequence:designator*
3. *sequence:designator* ,
4. *index:switch:list* , *index:switch:list*

Program:names must not be among the *sequence:designators* in a *switch::list*.

Index:switch:declaration means:

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

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

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

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

7. GOTO TOGGLE ($ Ø $) $
8. SWITCH CHOOSE = (, , STØ1 ($ ALPHA $), TOGGLE ($K$)) $

The *index* in a reference to an *index:switch* must be within the range indicated by the *switch:list* in the *declaration*. The *index* value points out the required *sequence:designator* according to its position in the list, starting with zero. *Commas* without corresponding *sequence:designators* indicate values of the *index* for which no transfer of control takes place. Thus, GOTO TOGGLE ($ Ø $) $ effects a transfer to BL97. If the reference to TOGGLE ($ K $) is activated; for K = 2 control is transferred to LOOP; for K = 3 control is transferred via *switch* EMIT -- where EMIT is an *item:switch* dependent on values of a *string:item* or two-dimensional *array*, in this case *bead* or *entry* ($ I, J $); 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.

5.32 *Item:switches*

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

1. *constant* = *sequence:designator*
2. *item:switch:list* , *item:switch:list*

Program:names must not be among the *sequence:designators* in <u>this</u> kind of *switch:list*, either.

Item:switch:declaration means one of the following expressions:

3. SWITCH *name* (*item:name*) = (*item:switch:list*) $
4. SWITCH *name* (*file:name*) = (*item:switch:list*) $

The *name* following the *primitive*, SWITCH, is the *switch:name*. The *item:switch:list* consists of *constants* paired with *sequence::designators*. 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 in accordance with the corresponding *sequence:designator*. 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:*

```
5.  SWITCH  WHICH (BETA) = (3H(ARY) = ST34, 3H(ØL9) = FINIS($A/2$),
    3H(   ) = SØ1, 3H(ABC) = SØ2, 3H(''') = EXIT, 3H(===) = SØ1,
    3H(.$.) = ESSO($A,B,C$), O(777777) = STØ1, 3H(XXX) = PCR'SORT) $
```

If a *file* is named in the *declaration*, the *constants* are *status: :constants* from the *file:declaration* representing conditions of the *file*. If the *switch:declaration* names a *file* or 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* (of the appropriate number of components) to select the particular *bead* or *entry* of the *item* to be compared with the *constants*. For example:

6. GOTO WHICH ($ J, K, L, M $) $

This *go:to:statement* implies that BETA (the *item* named in the *declaration* for WHICH) is a four-dimensional *array*.

5.4 *Closes*

Close:declaration means the following expression:

1. CLOSE *name* $ BEGIN *statement:list* 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*. 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 which 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. As with a *switch:declaration*, a *close:declaration* should not be placed in such a position among the *statements* of a *program* 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*, either directly or via a *switch*. 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*, which 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* (nor via a *switch:declaration*) outside that *loop: :statement*. A *close:declaration* outside the *loop:statement* should be invoked from within the *loop:statement* only if the *close:declaration* will not return control to the *loop:statement*. See section 3.76 on *processing:declarations* within *loop:statements* for more details.

5.5 *Procedures*

A *procedure:declaration* sets up a closed subroutine which 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. *data:declaration*
2. *program:declaration*
3. *declaration:list* *declaration:list*

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

4. *simple:item:name*
5. *array:name*
6. *table:name*
7. *close:name* .
8. *formal:input:parameter:list* , *formal:input:parameter:list*

Note that a *close:name* in a *formal:input:parameter:list* is followed by a *period*. In fact, it is the presence of the *period* in a *formal:input: :parameter:list* that defines the preceeding *name* as a *close:name*.

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

9. *simple:item:name*
10. *array:name*
11. *table:name*
12. *statement:name* .
13. *formal:output:parameter:list* , *formal:output:parameter:list*

A *statement:name* in a *formal:output:parameter:list* must be followed by a *period;* it is the presence of the *period* that defines it as a *statement::name.*

Procedure:heading means one of the following three expressions:

14. PROC *name* $ *optional:declaration:list*

15. PROC *name* (*optional:formal:input:parameter:list*) $ *optional:declaration:list*

16. PROC *name* (*optional:formal:input:parameter:list* = *formal:output:parameter:list*) $ *optional:declaration:list*

Procedure:body means

17. BEGIN *statement:list* END

Procedure:declaration means

18. *procedure:heading* *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* which calls it in turn, either directly or indirectly.

If the *procedure:heading* contains *formal:parameters* other than *close:names* and *statement:names*, they must be declared in the *procedure:declaration* before they are referenced in *statements*.

The *name* following PROC becomes the *procedure:name*. A *procedure:declaration* sets up a closed subroutine (or *procedure*) which is invoked by a *procedure::call:statement*. Normally, when execution of the *statement:list* is completed or a *return:statement* is executed, control returns to the *statement* which follows the invoking *procedure:call:statement*. If there is a *go:to:statement* or *switch* executed, which references a *statement:*

:name declared in the *formal:output:parameter:list*, control returns to the *statement* labelled with the corresponding *name* in the *actual::output:parameter:list*. Therefore, *output:parameter:statement:names* are called alternate exits. Under these circumstances, 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*. It is possible, however, for the *procedure* to contain a *go:to:statement* or *switch* which references a *statement:name* in the *main:program*. If control passes to that *main:program statement* through execution of such a *go:to:statement* or *switch*, then the final assignment process is bypassed and the *actual:output:parameters* corresponding to the *simple:item:names* among the *formal:output:parameters* are not changed. It is also possible that *loop:variables* in the *main:program* which were active at the time of calling the *procedure*, will not have their correct values. See section 3.55 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 *name* $ *optional:declaration:list*
2. PROC *name* (*optional:formal:input:parameter:list*) $ *optional:declaration:list*

Function:declaration means

3. *function:heading* *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 <u>same</u> *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* which acts as the *output:parameter* determines the type of *formula* represented by a *function:call*. The discussion in section 3.55 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*.

If the *function:heading* contains *formal:input:parameters* other than *close::names*, they must be declared in the *function:declaration* before they are referenced in *statements*.

Chapter 6. *Programs*

6.1 Other *Programs*

In section 5.2 we discussed the means whereby communication can be established between the present *program* and other *programs* compiled independently. Assumptions are made about the characteristics of the "other" *programs*, but there is no universal means for informing the compiler that the "present" *program* is to be compiled in a manner to make it a *program* of this "other" sort. For some compilers there may be compiler-dependent *declarations* or *directives* for accomplishing such a result.

If the compiler recognizes that this *program* is of this other sort, an independently compiled subroutine, it will most likely treat the *stop:statement* as an indication to return control to the external calling program.

6.2 The Present *Program*

The *program* means one of these two expressions:

1. START *statement:list* TERM $
2. START *statement:list* TERM *statement:name* $

In other words the *program* is a string of *statements*, *declarations*, and *directives* (see *statement:list* section 3.6) enclosed in the *brackets*, START and TERM, and followed by a *dollar:sign* or a *statement:name* and a *dollar:sign*.

If there is no *statement:name* following TERM, execution of the object program will begin with the first *statement* of the *main:program*. Otherwise, the *name* must be that of a *statement* of the *main:program* and execution will begin with that *named:statement*.

6.3 *Directives*

We have already discussed the *define:directive* (section 2.8) which makes it possible to direct the compiler to treat a *name* as an expression of one or more *symbols*. There is another *directive*, the *mode:directive*, which directs the compiler to change the mode for definition of otherwise undefined *names*. This is discussed in the next section.

6.4 *Mode:directives*

Mode:directive means one of these two expressions:

1. MODE *item:description* $
2. MODE *item:description* P *optionally:signed:constant* $

Each compiler assumes a normal mode for the definition of undeclared or otherwise undefined *simple:item:names*. The presence of a *mode::directive* causes the compiler to change the current mode to be in accordance with the *item:description*. If a pre-set value is also specified, all subsequent mode-defined *items* will be given this value initially.

The effect of a *mode:directive* begins at the point where it occurs among the *statements* and *declarations* of the *program* and lasts until the next *mode:directive* or the end of the *program* (TERM) is encountered. The new mode is established and persists irrespective of whether the *mode:directive* occurs in the *main:program* or a *procedure:* or *function:declaration*.

6.5 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. The designers of JOVIAL have catered 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 3.75, 3.76, and 3.77. 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* or *file*.

Following a *define:directive*, any occurrence of the *name* thereby defined will be effectively replaced by its definition, with these exceptions -- the *name* may be redefined by a new *define:directive*; there will be no replacement where the *name* occurs as part of a *status:constant*, *literal::constant*, or *comment*; the *name* will be 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*. The *names* fall into three categories as follows:

1. *device:names* (used only in *file:declarations*)
2. *statement:names*, *program:names*, *close:names*, *switch:names*
3. *item:names*, *table:names*, *file:names*, *procedure:names*, *function:names*

A *name* used in one of the above three categories may duplicate a *name* in one or both of the other categories without conflict or ambiguity.

There may even be duplication within a category 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* which 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* which is not part of a *procedure:* or *function:declaration* is the *main:program*. A *name* which is predefined or defined within the *main:program* is said to be "global."

Device:names are all predefined and there is no way to define them within the *program*. Therefore, *device:names* are always global.

All types of *names* in the above three categories may be predefined (by COMPOOL or otherwise). All *names* of categories 2 and 3 may be explicitly defined -- *statement:names* by being properly prefixed to *statements* or *clauses*; the others by *declarations*; and *statement:names* and *close:names*, locally, by appearing as *formal:parameters*.

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:* or *function:name* is both global and local to the *procedure:* or *function:declaration* which it names. One seeming conflict in naming is not only permitted but required -- a *function:declaration* must contain a *simple:item:declaration* duplicating 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* which do not have local definitions of the same *name* in the same category. Consider figure 9 for example. As *device:names*, T12 and T13 are predefined and global; their scope is the whole *program*. Assume that the other definitions are made explicitly as shown in the *main:program* or *procedures*. Then, with respect to figure 9A, T12 is:

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

With respect to figure 9B, T13 is:

5. as a *switch:name*, global; and its scope is only the *main:program*.
6. as a *procedure:name*, both global and local to *procedure* T13; its scope is the entire *program*.
7. as a *statement:name*, local to *procedure* T13.
8. 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*, *table:names*, and *file:names* which are to be defined by *declaration* must be declared before they are used in their respective scopes.

Definition by mode is possible only at points in the *program* for which there is no other definition, for the particular *name*, in category 3. Thus, any *name* which is predefined in category 3 cannot be redefined by mode. Otherwise, a *name* may be defined by mode as a global *simple:* *:item:name* only if there is no global *declaration* for that *name* in category 3. A *name* may be defined by mode as a local *simple:item:name* only if there is no category 3 *declaration* for that *name* in the same *procedure:* or *function:declaration* and no prior global category 3 definition for that *name* (either by mode or by *declaration*).

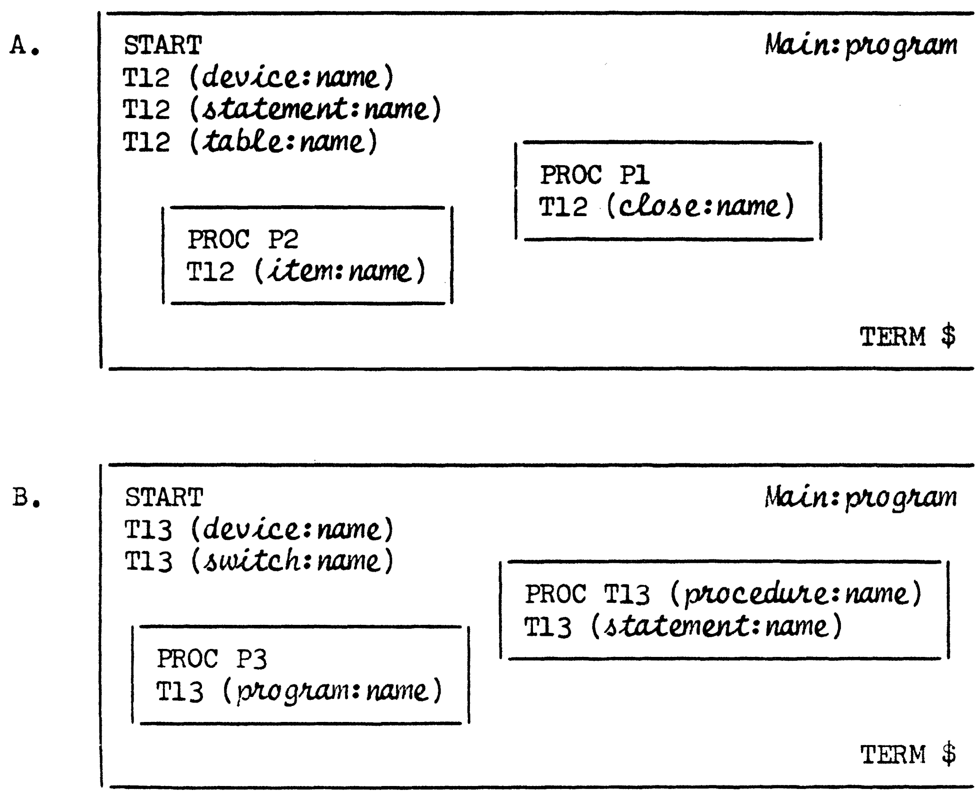

Figure 9. Scope

Index and Glossary

Three kinds of words are indexed below: English or programmer's jargon, metalanguage words and phrases, and JOVIAL *primitives*. Two kinds of references are given: numbers of sections where the word or phrase is used and numbers of sections where it is defined. Defining section numbers are typed in script.* For example, see the entry for *abbreviation* below. The term is defined in section *2.5*; it is used in sections 2.4, 2.5, 2.7, 4.41, and 4.6. For the main index listing of metalanguage words and phrases, all references are indexed. For English words or jargon and for subordinate listings of metalanguage words and phrases, usually only defining references are indexed. For *primitives* all references are indexed. Since these words <u>are</u> *primitives* there can be no definitions.

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. Others should consult the defining section to avoid overlooking important exceptions and qualifying remarks. It is also necessary to be familiar with the rules concerning the use of *spaces* as explained in section 2.2. 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 which is merely a continuation of the previous line is not numbered.

* Defining section numbers are also underlined

1. *actual:output:parameter:list*	1.2	<u>3.55</u>	5.5
2. *actual:output:parameter*			
3. *actual:output:parameter* , *actual:output: :parameter:list*			
4. *actual:parameter*	3.31	3.55	3.75
5. *actual:input:parameter*			
6. *actual:output:parameter*			
7. *actual:parameter:formula*			3.55
8. *formula* in *actual:input:parameter:list*			
9. *actual:parameter:list*		1.2	3.55
10. *actual:input:parameter:list*			<u>3.55</u>
11. *actual:output:parameter:list*			<u>3.55</u>
12. *actual:parameter:name*			3.55
13. Any of the following *names* when occurring in an *actual:parameter:list*. In the reference in section 3.55 *item:name* is obviously not included:			
13. *array:name*			<u>4.43</u>
14. *close:name*			<u>5.4</u>
15. *item:name*			
16. *statement:name*			<u>3.4</u>
17. *table:name*			
18. *actual:parameter:table*			3.55
19. a *table*, the *name* of which occurs in an *actual:parameter:list*			
20. *actual:parameter:variable*			3.55
21. *variable* in *actual:parameter:list*			
22. ALL		2.5	3.74
23. alternate exit		3.75	<u>5.5</u>
24. *output:parameter:statement:name*			
25. *statement:name* in *output:parameter:list*			
26. *alternative*			<u>3.73</u>
27. *or:if:clause* *independent:statement*			
28. *statement:name* . *alternative*			
29. *alternative:list*			<u>3.73</u>
30. *if:either:clause* *independent:statement* *alternative*			
31. *alternative:list* *alternative*			
32. *alternative:statement*	3.7	<u>3.73</u>	5.1
33. *alternative:list* END			
34. AND		2.5	3.36

No.	Entry	References
1.	*arithmetic:operator*	2.4 3.33 3.34 3.36
2.	+	
3.	-	
4.	*	
5.	/	
6.	**	
7.	ARRAY	2.5 4.52 4.53
8.	*array*	3.1 3.22 3.33 3.56 4.43 4.5 4.51 4.52 4.54 5.31 5.32
9.	collection of data declared by an *array::declaration*	
10.	essentially equivalent to *array:item*, but *array* has the connotation of the group of data	
11.	*array:declaration*	3.21 4.52
12.	ARRAY *name dimension:list item:description* $	
13.	ARRAY *name dimension:list item:description* $ *constant:list*	
13.	specification of a one or more dimensional, rectangular array of similar data values	
14.	*array:item*	3.33 4.52
15.	collection of data declared by an *array:declaration*	
16.	essentially equivalent to *array*, but *array:item* has the connotation of a member of the group	
17.	*array:item:declaration*	4.4
18.	*array:declaration*	4.52
19.	*array:item:name*	4.43 4.52
20.	*array:name*	4.43
21.	*name* following ARRAY in an *array:declaration*	
22.	*array:name*	3.55 3.56 4.43 5.5
23.	*array:item:name*	4.52
24.	*name* following ARRAY in an *array:declaration*	

1. ASSIGN 2.5 3.71
2. *assignment:statement* 3.1 3.24 3.5 3.51 3.52 3.55 3.56 3.71 3.73 4.42 4.56 6.5
3. *variable* = *formula* $
4. *statement* specifying that the value of a *variable* be changed to the current value of a *formula*. The *variable* and *formula* must be of compatible data types.
5. *basic:structure* 4.59
6. property of a *table* being parallel or serial. Serial means the words of an *entry* occupy a contiguous block. Parallel means the words of an *entry* are similarly placed in separate blocks.
7. *basic:structure:specification* 4.56 4.58 4.59
8. P for parallel
9. S for serial
10. part of a *table:declaration*
11. *bead* 4.5 4.57 4.58 5.31 5.32
12. a particular occurrence of a *string:item*, specified by a two-component *index*, the first component indicating which *bead* within the *entry*, the second component indicating which *entry* of the *table*.
13. BEGIN 2.5 2.7 3.4 3.6 3.72 3.73 3.74 4.51 4.52 4.53 4.56 4.58 5.5
14. *binary:file* 4.6
15. *file*, declared with B following the *file:* *:name*, in which data are represented with the same bit patterns that are used in the internal memory of the computer
16. BIT 2.5 3.24 3.26
17. bit 2.61
18. binary digit
19. Ø or 1
20. basic unit of information

1. *blank* 2.3 3.35 3.51
2. *space*
3. the JOVIAL character represented with no ink on the paper
4. *boolean* 2.61 3.2 3.23 3.31
5. pertaining to the algebra of truth values
6. having one of two possible values, "true" or "false," represented by 1 and Ø respectively.
7. *boolean:constant* 2.63 3.36
8. Ø
9. 1
10. *boolean:formula* 3.35 3.36 3.51 3.72 3.73 3.77 4.6
11. *boolean:constant* 2.63
12. *boolean:function*
13. *boolean:variable* 3.27
14. *relational* proposition
15. combination of *boolean:formulas* with *parentheses* and *boolean:operators*
16. *boolean:function* 3.36
17. invocation of a *function:declaration* with a *boolean* output value.
18. *boolean:item* 4.42 4.52
19. *item* specified by *declaration* in which B follows the *item:name*
20. *boolean:item:description* 4.41
21. B
22. *boolean:operator* 3.36
23. AND
24. NOT
25. OR
26. *boolean:variable* 2.61 3.27 3.36 3.51 3.52
27. *boolean:item*
28. ODD (*loop:variable*)
29. ODD (*named:numeric:variable*)
30. *bracket* 2.4 3.33 3.34 3.6 6.2
31. BEGIN END
32. DIRECT JOVIAL
33. IFEITH END
34. START TERM
35. ( )
36. (/ /)
37. ($ $)
38. (* *)
39. '' ''

1. BYTE 2.5 3.26
2. byte 3.26
3. computer representation of one character of a *hollerith* or *transmission:code* value
4. CHAR 2.5 3.24
5. characteristic
6. integral part of a logarithm
7. exrad, by analogy with logarithms
8. *clause* 3.72 4.1 6.5
9. *for:clause*
10. *complete:for:clause* 3.74
11. *incomplete:for:clause* 3.74
12. *if:clause* 3.72 3.73
13. *if:either:clause* 3.73
14. *or:if:clause* 3.73
15. CLOSE 2.5 5.4
16. *close* 5.4 5.5
17. *close:declaration* 5.4
18. *close:declaration* 3.53 3.54 3.75 3.76 5.1 5.4
19. CLOSE *name* $ *independent:statement*
20. a subroutine without parameters, sensitive to the scope of definition of *for:variables*
21. *close:name* 3.55 3.75 3.76 4.1 5.4 5.5 5.6 6.5
22. the *name* in a *close:declaration* following the *primitive* CLOSE
23. *comma* 2.3 3.22 4.43 4.55 5.31
24. ,
25. *comment* 2.2 2.4 2.5 2.7 2.8 6.5
26. '' *signs* ''
27. equivalent to *space* in most places
28. *complete:for:clause* 3.74 3.77
29. FOR *loop:variable* = *numeric:formula*, *numeric:formula*, *numeric:formula* $
30. FOR *loop:variable* = ALL (*table:name*) $
31. FOR *loop:variable* = ALL (*table:item:name*) $
32. *complete:loop:statement* 3.74 3.77
33. *complete:for:clause* *independent:statement*
34. *complete:for:clause* *special:compound*
35. *complete:for:clause* *incomplete:loop:statement*

1.	*complex*							3.6
2.	pertaining to a *complex:statement*							
3.	*complex:statement*				2.1	3.4	3.7	3.71
4.	*alternative:statement*							3.73
5.	*conditional:statement*							3.72
6.	*direct:statement*							3.71
7.	*loop:statement*							3.74
8.	COMPOOL							4.2
9.	communication pool							
10.	a table or dictionary of system definitions							
11.	*compound*							3.6
12.	pertaining to a *compound:statement*							
13.	*compound:statement*				2.1	3.4	3.6	3.72
14.	BEGIN *statement:list* END							
15.	*conditional:statement*				3.7	3.72	3.73	5.1
16.	*if:clause independent:statement*							
17.	*constant*	2.4	2.6	2.61	2.63	2.7	3.1	3.28
		3.3	3.56	3.71	3.73	4.41	4.42	4.43
		4.51	4.53	4.55	4.57	5.2	5.32	6.4
18.	*boolean:constant*							2.63
19.	*dual:constant*							2.63
20.	*literal:constant*							
21.	*hollerith:constant*							2.63
22.	*octal:constant*							2.63
23.	*transmission:code:constant*							2.63
24.	*numeric:constant*							
25.	*fixed:constant*							2.63
26.	*floating:constant*							2.63
27.	*integer:constant*							2.63
28.	*octal:constant*							2.63
29.	*status:constant*							2.63
30.	*constant:list*				4.51	4.52	4.53	4.55
31.	dimensioned list of *constants* for presetting values of *table:items*, *strings*, and *arrays*							
32.	*one:dimensional:constant:list*							4.51
33.	*two:dimensional:constant:list*							
34.	etc.							4.51
35.	*data:declaration*			2.1	4.2	4.4	5.2	5.5
36.	*file:declaration*							4.6
37.	*item:declaration*							4.4
38.	*indexed:item:declaration*							4.4
39.	*array:item:declaration*							4.52
40.	*string:item:declaration*							4.57
41.	*table:item:declaration*							
42.	*simple:item:declaration*							4.42
43.	*overlay:declaration*							
44.	*table:declaration*							4.54

1. *data:sequence* 4.43 4.55
2. *independent:data:sequence* 4.43
3. *subordinate:data:sequence* 4.55
4. part of *overlay:declaration*
5. *decimal:point* 2.3 2.7
6. .
7. *declaration* 2.1 3.1 3.2 3.21 3.24 3.51 3.54 3.55 3.6 3.76 4.1 4.4 4.54 4.55 4.57 4.59 5.2 5.31 5.32 5.4 6.1 6.2 6.4 6.5
8. *data:declaration* 4.4
9. *processing:declaration* 5.1
10. *declaration:list* 5.5 5.6
11. *data:declaration* 4.4
12. *program:declaration* 5.2
13. *declaration:list* *declaration:list*
14. DEFINE 2.5 2.8
15. *define:directive* 2.7 2.8 4.6 6.3 6.5
16. DEFINE *name* ''*signs*'' $
17. defined 2.8 6.5
18. with respect to *loop:variable:* within its active range
19. with respect to *name*: given a meaning within a scope within a *program*
20. *defined:entry* 4.57
21. *table:entry* for which the location of each *item* is specified by the programmer.
22. *defined:entry:description* 4.57 4.58
23. *defined:entry:item:declaration* 4.57
24. *string:item:declaration* 4.57
25. *defined:entry:description* *defined:entry:description*
26. *defined:entry:item:declaration* 4.57
27. *defined:entry:table* 4.58
28. *table* declared by a *defined:entry:table:declaration*
29. *defined:entry:table:declaration* 4.54 4.58
30. *description* 4.41 4.42 4.43
31. *item:description*
32. *device:name* 4.6 6.5
33. a compiler-dependent *name* permanently assigned to an input-output device for which a *file* may be declared
34. *dimension:list* 4.52
35. a string of *numbers*; part of an *array:declaration*
36. DIRECT 2.5 3.71

1. *direct:assign* 2.8 3.71 6.5
2. ASSIGN A(*signed:number*) = *named:variable* $
3. ASSIGN *named:variable* = A(*signed:number*) $
4. *direct:code* 2.8 3.71 6.5
5. *signs* 2.3
6. *direct:assign* 3.71
7. *direct:code* *direct:code*
8. *direct:statement* 3.7 3.71
9. DIRECT *direct:code* JOVIAL
10. *directive* 2.1 3.6 6.1 6.2 6.3
11. *define:directive* 2.8
12. *mode:directive* 6.4
13. *dollar:sign* 2.3 3.4 3.54 4.59 6.2
14. $
15. *dual* 2.61 3.2 3.23 3.34 3.51 4.52
16. pertaining to data forms and values having two components
17. *dual:array* 4.52
array whose *declaration* contains a *dual:item:description*
18. *dual:assignment:statement* 3.51
dual:item = *dual:formula* $
19. *dual:constant* 2.63 3.34 4.41
20. *dual:exchange:statement* 3.52
21. *dual:item* == *dual:item* $
22. *statement* specifying that the values of two *dual:items* be exchanged
23. *dual:formula* 3.1 3.34 3.35 3.36 3.51
24. *numeric:formula* 3.33
25. *dual:constant* 2.63
26. *dual:item*
27. *dual:function*
28. arithmetic combinations of *dual:formulas*
29. *dual:function* 3.34
30. invocation of a *function:declaration* with a *dual* output value
31. *dual:item* 4.41
32. *item* specified by *declaration* in which the *item:name* is followed by D or a *dual:constant*
33. *dual:item:description* 4.41
34. *dual:relation:list* 3.35 3.36
35. list of *relational:operators* and *dual:formulas*
36. *dual:specifier* 4.41

1.	*dual:variable*					3.34	3.51	3.52
2.	*dual:item*							
3.	END	2.5	3.4	3.6	3.72	3.73	3.74	3.77
			4.51	4.52	4.53	4.56	4.58	5.5
4.	ENT					2.5	3.28	3.52
5.	ENTRY						2.5	3.28
6.	*entry*	3.22	3.24	3.28	3.33	3.35	3.52	3.56
		4.43	4.5	4.55	4.56	4.57	4.58	4.59
							5.31	5.32
7.	one occurrence of an *array:item*; which one specified by an *index*							
8.	the set of all the *items* of a *table* with the same second component of the *index* in the case of *strings* and the same *index* in the case of other *items*							
9.	*entry:assignment:statement*							3.51
10.	*entry:variable* = *entry:formula* $							
11.	*statement* specifying that the value of an *entry:variable* be changed to the current value of an *entry:formula*							
12.	*entry:description*					4.55	4.56	4.58
13.	*defined:entry:description*							4.57
14.	*ordinary:entry:description*							4.55
15.	*entry:formula*				3.32	3.35	3.36	3.51
16.	∅							
17.	*entry:variable*							3.28
18.	*entry:index*							3.56
19.	*index* used to specify which *entry* of *array* or *table*							
20.	*entry:variable*			3.28	3.32	3.36	3.51	3.52
21.	ENT (*table:name* ($ *index* $))							
22.	ENT (*table:item:name* ($ *index* $))							
23.	ENTRY (*table:name* ($ *index* $))							
24.	ENTRY (*table:item:name* ($ *index* $))							
25.	EQ	2.5	2.7	3.35	3.36	3.4	3.74	4.58
26.	*equals:sign*					2.3	4.43	4.55
27.	=							
28.	*exchange:statement*						3.5	3.52
	variable == *variable* $							
29.	express							
30.	global							
31.	exrad							2.61
32.	exponent of the radix in floating representations of numbers							

1.	FILE	2.5 4.6
2.	*file*	2.6 3.6 3.24 *3.56* 3.57 3.58 4.2 4.6 5.32 6.5
3.	a collection of *records* on an input or output device	
4.	*file:declaration*	4.4 *4.6* 5.32 6.5
5.	*file:name*	2.63 3.1 3.24 3.36 3.56 3.57 3.58 *4.6* 5.32 6.5
6.	*name* following FILE in a *file:declaration*	
7.	*file:structure:specification*	*4.6*
8.	H *number* V *number*	
9.	B *number* V *number*	
10.	H *number* R *number*	
11.	B *number* R *number*	
12.	*fixed*	*2.61* 3.2 3.33 3.51 3.71 4.41
13.	pertaining to values with a specified number (which may even be zero or negative) of bits after the binary point	
14.	*fixed:constant*	*2.63* 4.53
15.	*fixed:item*	4.53
16.	*item* specified by *declaration* in which the *item:name* is followed by a *fixed:item:description* or a *fixed:constant*	
17.	*fixed:item:description*	*4.41*
18.	*fixed:specifier*	*4.41*
19.	*fixed:variable*	2.61 3.2 *3.25*
20.	*fixed:item*	
21.	MANT (*floating:item*)	
22.	*floating*	*2.61* 3.2 3.23 3.31 3.33 3.51 3.71 4.41

pertaining to values (v in the equation below) represented by two numbers (s for signicand and e for exrad in the equation below)

$$v = s \times 2^{e}$$

where $s = \emptyset$ or $|s| = 1/2$ or $1/2 < |s| < 1$

1.	*floating:constant*			2.61	2.63	4.41
2.	*floating:item*				4.41	4.53
3.	*item* specified by *declaration* in which the *item:name* is followed by F or a *floating:constant*					
4.	*floating:item:description*					4.41
5.	floating-point					
6.	*floating*					2.61
7.	*floating:variable*			3.2	3.24	3.25
8.	*floating:item*					
9.	FOR		2.5	3.74	3.75	4.58
10.	*for:clause*		3.74	3.75	3.76	3.77
11.	*complete:for:clause*					3.74
12.	*incomplete:for:clause*					3.74
13.	for-variable					2.5
14.	*loop:variable*				2.5	3.74
15.	*formal:input:parameter*			3.55	5.6	6.5
16.	*array:name*					4.43
17.	*close:name* .					
18.	*simple:item:name*					
19.	*table:name*					
20.	*formal:input:parameter:list*		1.2	4.1	5.5	5.6
21.	*formal:input:parameter*					
22.	*formal:input:parameter* , *formal:input:parameter:list*					
23.	*formal:output:parameter*			3.55	5.5	6.5
24.	*array:name*					4.43
25.	*simple:item:name*					
26.	*statement:name* .					
27.	*table:name*					
28.	*formal:output:parameter:list*		1.2	4.1	5.5	5.6
29.	*formal:output:parameter*					
30.	*formal:output:parameter* , *formal:output:parameter:list*					
31.	*formal:parameter*	3.31	3.55	4.3	5.5	6.5
32.	*formal:input:parameter*					
33.	*formal:output:parameter*					
34.	*formal:parameter:item*					3.55
35.	an *item*, the *name* of which occurs in a *formal:parameter:list*. In the reference in section 3.55, obviously only *simple:items* are meant					

1.	*formal:parameter:list*							1.2
2.	*formal:input:parameter:list*							
3.	*formal:output:parameter:list*							
4.	*formal:parameter:table*							3.55
5.	a *table*, the *name* of which occurs in a *formal:parameter:list*							
6.	*formula*	3.1	3.22	3.3	3.33	3.35	3.36	3.51
					3.55	3.74	3.75	5.6
7.	*boolean:fomula*							3.36
8.	*dual:formula*							3.34
9.	*entry:formula*							3.32
10.	*literal:formula*							
11.	*hollerith:formula*							3.32
12.	*transmission:code:formula*							3.32
13.	*numeric:formula*							3.33
14.	*status:formula*							3.32
15.	*function*	2.6	3.1	3.31	3.3	3.36	3.51	3.75
					4.2	5.4	5.5	5.6
16.	invocation of a *function:declaration* by *name*							
17.	*function:declaration* (context will make it clear when this usage is intended)							
18.	*function:call*			3.31	3.76	4.1	5.5	5.6
19.	*function*							3.31
20.	*function:declaration*	3.31	3.51	3.54	3.75	3.76	4.1	4.2
			5.1	5.4	5.5	5.6	6.4	6.5
21.	*function:heading*							5.6
22.	*function:name*						3.31	6.5
23.	*name* following PROC in a *function:declaration*							
24.	*functional:modifier*			2.4	3.1	3.24	3.25	3.27
25.	ABS							
26.	ALL							
27.	BIT							
28.	BYTE							
29.	CHAR							
30.	ENT							
31.	ENTRY							
32.	'LOC							
33.	MANT							
34.	NENT							
35.	NWDSEN							
36.	ODD							
37.	POS							

No.	Entry							
1.	global						4.3	6.5
2.	defined within the *main:program* and those *procedures* without a conflicting local definition							
3.	*go:to:statement*	3.5	3.53	3.54	3.75	3.77	5.1	5.2
			5.3	5.31	5.32	5.4	5.5	6.5
4.	GOTO *name* $							
5.	GOTO *name* ($ *index* $)$							
6.	directs the sequence of *statement* executions elsewhere							
7.	GOTO	2.5	2.7	3.53	3.6	3.72	3.73	3.74
						5.3	5.31	5.32
8.	GQ						2.5	3.35
9.	GR			2.5	3.35	3.36	3.73	3.74
10.	*hollerith*	2.61	2.63	3.2	3.23	3.26	3.31	3.35
						3.51	3.58	4.6
11.	pertaining to the computer-dependent internal encoding of *signs*; the normal encoding scheme for the computer							
12.	*hollerith:constant*				2.2	2.61	2.63	3.32
13.	*number*H(*signs*)							
14.	the number of *signs* is *number*							
15.	*hollerith:file*							4.6
16.	*file*, declared with H following the *file:name*, in which data are represented as strings of *signs* coded in *hollerith*							
17.	*hollerith:formula*							3.32
18.	*hollerith:constant*							2.63
19.	*hollerith:function*							
20.	*hollerith:variable*							
21.	*octal:constant*							2.63
22.	*hollerith:function*							3.32
23.	invocation of a *function:declaration* with a *hollerith* output value							
24.	*hollerith:item:description*							4.41
25.	**H** *number*							
26.	*hollerith:variable*					3.2	3.32	4.6
27.	*hollerith:item*							
28.	*item* specified by *item:declaration* in which H follows the *item:name*							
29.	BYTE ($ *index* $) (*hollerith:item*)							
30.	*ideogram*		2.4	2.5	2.7	3.33	3.56	4.41
31.	IF			2.5	2.7	3.4	3.72	4.58
32.	*if:clause*				3.72	3.73	3.74	3.77
33.	IF *boolean:formula* $							
34.	*statement:name* . *if:clause*							

1\. *if:either:clause* 3.73
2\. IFEITH *boolean:formula* $
3\. IFEITH 2.5 3.73
4\. *incomplete:for:clause* 3.74
5\. FOR *loop:variable* = *numeric:formula* $
6\. FOR *loop:variable* = *numeric:formula* , *numeric:formula* $
7\. *incomplete:loop:statement* 3.74 3.77
8\. *loop:statement* headed by *incomplete:for:clauses*
9\. *independent:data:sequence* 4.43
10\. a string of *simple:item:names*, *table:names*,
and *array:names* separated by *commas*
11\. part of an *independent:overlay:declaration*
12\. *independent:overlay* 4.43
13\. the arrangement of *tables*, *arrays*, and
simple:items specified by an
independent:overlay:declaration
14\. *independent:overlay:declaration* 4.43 4.55 4.56
15\. *independent:overlay:specification* 4.43
16\. a string of *independent:data:sequences*
separated by *equals:signs*
17\. part of an *independent:overlay:declaration*
18\. *independent:statement* 3.4 3.72 3.73 3.74 3.75
19\. *compound:statement* 3.6
20\. BEGIN *statement:list* END
21\. *simple:statement* 3.5
22\. *index* 3.22 3.24 3.26 3.28 3.53 3.56 4.5
4.52 4.55 4.57 4.58 5.3 5.31 5.32
23\. *numeric:formula* 3.33
24\. *index* , *numeric:formula*
25\. *index:switch* 5.31
26\. *switch* specified by an *index:switch:declaration*
27\. *index:switch:declaration* 5.3 5.31
28\. SWITCH *switch:name* = (*index:switch:list*) $
29\. *index:switch:list* 5.31

1. *indexed:item*			4.53	5.32
2. *array:item*				
3. *table:item*				
4. *defined:entry:item*				
5. *ordinary:table:item*				
6. *string:item*				
7. *indexed:item:declaration*				*4.4*
8. *array:item:declaration*				
9. *table:item:declaration*				
10. *indexed:variable*	3.2	*3.22*	3.23	3.24
11. *name* ($ *index* $)				
12. *indexed:item*				
13. INPUT			2.5	3.57
14. *input:operand*			*3.56*	3.57
15. *array:name*				*4.43*
16. *table:name*				
17. *table:name* ($ *index* $)				
18. *table:name* ($ *index* ... *index* $)				
19. *variable*				

No.	Entry	References
1.	*input:parameter*	3.76 5.5 5.6
2.	*actual:input:parameter*	
3.	*formal:input:parameter*	
4.	the values and structures, specified or to be specified, for a *procedure:declaration* or *function:declaration* to work with	
5.	*input:parameter:list*	1.2 3.55
6.	*actual:input:parameter:list*	3.55
7.	*formal:input:parameter:list*	5.5
8.	*input:statement*	3.5 3.56 3.57 3.58 4.56
9.	INPUT *file:name input:operand* $	
10.	*integer*	2.61 3.2 3.23 3.33 3.51 4.41
11.	a whole number	
12.	having whole number values	
13.	*integer:constant*	2.63 3.71 4.41 5.2
14.	*integer:item*	4.42 4.53
15.	*item* specified by *declaration* in which the *item:name* is followed by an *integer:item:description* or an *integer:constant*	
16.	*integer:item:description*	4.41
17.	*integer:specifier*	4.41
18.	*integer:variable*	3.2 3.24 3.33 3.74 3.75
19.	*integer:item*	
20.	*loop:variable*	2.5
21.	BIT ($ *index* $) (*item*)	
22.	CHAR (*floating:item*)	
23.	POS (*file:name*)	
24.	NENT (*name*)	
25.	ITEM	2.5 4.42 4.55 4.56 4.57
26.	*item*	3.1 3.2 3.21 3.22 3.24 3.26 3.28 3.33 3.55 3.74 4.2 4.3 4.41 4.42 4.43 4.5 4.53 4.55 4.56 4.57 4.58 4.59 5.3 5.32 5.6 6.4
27.	*item* may be subdivided in two independent ways shown in groups 1 and 2 below. Even finer division is possible by choosing adjectives from both groups as in *simple:boolean:item*	
28.	Group 1	
29.	*boolean:item*	
30.	*dual:item*	

1.	*literal:item*							
2.	*hollerith:item*							
3.	*transmission:code:item*							
4.	*numeric:item*							
5.	*fixed:item*							
6.	*floating:item*							
7.	*integer:item*							
8.	*status:item*							
9.	Group 2							
10.	*indexed:item*							
11.	*array:item*							
12.	*table:item*							
13.	*defined:entry:item*							
14.	*ordinary:table:item*							
15.	*string:item*							
16.	*simple:item*							
17.	data structure specified by an *item:declaration*							
18.	*item:declaration*			3.2	3.21	4.4	4.41	4.55
19.	*indexed:item:declaration*							4.4
20.	*array:item:declaration*							4.52
21.	*table:item:declaration*							
22.	*defined:entry:item:declaration*							4.57
23.	*ordinary:table:item:declaration*							4.55
24.	*string:item:declaration*							4.57
25.	*simple:item:declaration*							4.42
27.	*item:description*	4.41	4.42	4.52	4.53	4.55	4.57	6.4
28.	*boolean:item:description*							4.41
29.	B							
30.	*dual:item:description*							4.41
31.	*fixed:item:description*							4.41
32.	*floating:item:description*							4.41
33.	*hollerith:item:description*							4.41
34.	H *number*							
35.	*integer:item:description*							4.41
36.	*transmission:code:item:description*							4.41
37.	T *number*							
38.	part of an *item:declaration*							
39.	*item:name*	2.63	3.1	3.33	3.55	4.5	4.55	4.57
					4.59	4.6	5.32	6.5
40.	*name* following ARRAY or ITEM or STRING in an *item:declaration*							

1. *item:switch* 5.31 5.32
2. *switch* specified by an *item:switch:declaration*
3. *item:switch:declaration* 5.3 5.32
4. SWITCH *switch:name*
   (*file:name*) = (*item:switch:list*) $
5. SWITCH *switch:name*
   (*item:name*) = (*item:switch:list*) $
6. *item:switch:list* 5.32
7. *constant* = *sequence:designator*
8. *item:switch:list* , *item:switch:list*
9. JOVIAL 2.5 3.71
10. *k:dimensional:constant:list* 4.51
11. BEGIN string of *k:minus:one:dimensional:*
    :constant:lists END
12. *k:plus:one:dimensional:constant:list* 4.51
13. BEGIN string of *k:dimensional:constant:*
    :lists END
14. *left:parenthesis* 2.3 2.4 2.63 3.71
15. (
16. *letter* 1.2 2.3 2.5 2.62 2.63 3.74 4.56 4.59 6.5
17. library 4.2
18. a collection of subroutines which may
    be incorporated in new *programs*
19. *like:table* 4.59
20. *table* declared by *like:table:declaration*
21. *like:table:declaration* 4.54 4.59
22. *list* 3.55 4.6
23. *constant:list*
24. *parameter:list*
25. *statement:list* 3.6
26. *status:list* 4.6
27. etc.
28. *literal* 2.61 2.63 3.35 3.52
29. *hollerith* 2.61
30. *octal*, depending on context
31. *transmission:code* 2.61
32. *literal:assignment:statement* 3.51
33. *literal:variable* = *literal:formula* $
34. *literal:constant* 2.7 2.8 6.5
35. *hollerith:constant* 2.63
36. *octal:constant* 2.63
37. *transmission:code:constant* 2.63

1.	*literal:formula*				3.32	3.35	3.36	3.51
2.	*hollerith:formula*							3.32
3.	*transmission:code:formula*							3.32
4.	*literal:item*						3.26	4.43
5.	*item* specified by *item:declaration* in which the *item:name* is followed by H (for *hollerith*) or T (for *transmission:code*)							
6.	*literal:relation:list*						3.35	3.36
7.	list of *relational:operators* and *literal:formulas*							
8.	*literal:variable*				3.2	3.26	3.51	3.52
9.	*hollerith:variable*							
10.	*transmission:code:variable*							
11.	'LOC					2.5	2.7	3.33
12.	local						4.3	6.5
13.	defined only within a *procedure*							
14.	*logical:operator*						2.4	3.36
15.	AND							
16.	NOT							
17.	OR							
18.	*loop:statement*	3.54	3.7	3.74	3.75	3.76	3.77	4.58
				5.1	5.3	5.4	5.5	5.6
19.	a string of *for:clauses* followed by an *independent:statement* or *special:compound*							
20.	*loop:variable*	2.4	2.5	2.63	2.7	3.24	3.27	3.54
		3.74	3.75	3.76	3.77	5.2	5.5	6.5
21.	*letter* following FOR in a *for:clause*							
22.	LQ						2.5	3.35
23.	LS			2.5	3.35	3.36	3.72	3.73
24.	*main:program*		4.3	5.1	5.5	6.2	6.4	6.5
25.	all of the *program* which is not part of any *procedure:declaration* or *function:declaration*							
26.	MANT						2.5	3.25
27.	mantissa							
28.	fractional part of a logarithm							
29.	signicand, by analogy with logarithms							
30.	*mark*						2.3	2.5

1.	metalanguage		1.1	1.2
2.	a mode of expression which transcends language			
3.	a language used to explain or describe another language			
4.	*minus:sign*			2.3
5.	-			
6.	MODE		2.5	6.4
7.	*mode:directive*	3.51	6.3	6.4
8.	MODE *item:description* $			
9.	MODE *item:description* P *constant* $			
10.	*modifier*	3.24	3.26	3.74
11.	*functional:modifier*			
12.	*n1n*	4.4	4.56	4.58
13.	a *number*			2.62
14.	the (maximum) number of *entries* specified for a *table*			
15.	*n2n*	4.4	4.56	4.58
16.	a *number*			2.62
17.	the nominal number of words per *entry* of a *table*			
18.	*n3n*	4.4	4.57	4.58
19.	a *number*			2.62
20.	the index of the word of the *entry* containing an *item*, or in which the *item* begins			
21.	*n4n*		4.4	4.57
22.	a *number*			2.62
23.	the index of the bit of the word in which an *item* begins			
24.	*n5n*		4.4	4.57
25.	a *number*			2.62
26.	the increment from word to word of an *entry* containing *beads* of a *string:item*			
27.	*n6n*		4.4	4.57
28.	a *number*			2.62
29.	the number of *beads* in each of the words of an *entry* containing *beads* of a *string:item*			
30.	*n7n*	4.4	4.41	4.43
31.	a *number*			2.62
32.	the number of bits or bytes specified for an *item* or for each component of an *item*			

1.	*n8n*	4.4	4.41
2.	a *number*		2.62
3.	the number of fractional bits specified for an *item* or for each component of an *item*		
4.	*n9n*	4.4	4.6
5.	a *number*		2.62
6.	the estimated maximum number of *records* in a *file*		

1.	*n1Øn*							4.4	4.6
2.	a *number*								2.62
3.	the (estimated maximum) number of bits or bytes in a *record*								
4.	*name*	2.4	2.5	2.63	2.7	2.8	3.1	3.21	3.22
		3.24	3.28	3.31	3.33	3.4	3.53	3.55	3.73
		3.74	3.75	3.76	3.77	4.1	4.2	4.3	4.4
		4.42	4.43	4.52	4.54	4.55	4.56	4.57	4.58
		4.59	4.6	5.1	5.2	5.31	5.32	5.4	5.5
		5.6	6.2	6.3	6.5				
5.	*named:boolean:variable*								3.27
6.	*boolean:item*								
7.	*named:fixed:variable*								3.25
8.	*fixed:item*								
9.	*named:integer:variable*								3.24
10.	*integer:item*								
11.	*named:literal:variable*								3.26
12.	*literal:item*								
13.	*named:numeric:variable*								3.27
14.	*fixed:item*								
15.	*floating:item*								
16.	*integer:item*								
17.	*named:statement*						3.33	3.4	6.2
18.	*name* . *statement*								
19.	*named:variable*				3.2	3.23	3.24	3.26	3.71
20.	*boolean:item*								
21.	*dual:item*								
22.	*fixed:item*								
23.	*floating:item*								
24.	*integer:item*								
25.	*literal:item*								
26.	*status:item*								
27.	NENT			2.5	3.24	3.33	3.74	4.56	4.58
28.	NOT							2.5	3.36
29.	NQ						2.5	3.35	3.36
30.	*number*	2.62	2.63	4.4	4.41	4.43	4.52	4.6	5.2
31.	a string of *numerals*								

1.	*numeral*	2.3 2.5 2.62 2.63 4.59 6.5
2.	Ø	
3.	1	
4.	2	
5.	3	
6.	4	
7.	5	
8.	6	
9.	7	
10.	8	
11.	9	
12.	*numeric*	2.61 2.63 3.33 3.34 3.35 3.52 3.71 4.57
13.	*fixed*	2.61
14.	*floating*	2.61
15.	*integer*	2.61
16.	*octal*, depending on context	
17.	*numeric:assignment:statement*	3.51
18.	*numeric:variable* = *numeric:formula* $	
19.	*numeric:constant*	3.33
20.	*fixed:constant*	2.63
21.	*floating:constant*	2.63
22.	*integer:constant*	2.63
23.	*octal:constant*	2.63
24.	*numeric:formula*	3.1 3.22 3.33 3.34 3.35 3.36 3.51 3.74 3.75 4.56 4.57
25.	*numeric:function*	3.33
26.	invocation of a *function:declaration* with a *numeric* output value	
27.	*numeric:relation:list*	3.35 3.36
28.	list of *relational:operators* and *numeric:formulas*	
29.	*numeric:variable*	3.2 3.33 3.51 3.52 4.56
30.	*fixed:variable*	3.25
31.	*fixed:item*	
32.	MANT (*floating:item*)	
33.	*floating:variable*	
34.	*floating:item*	
35.	*integer:variable*	3.24
36.	*integer:item*	
37.	*loop:variable*	2.5 3.74
38.	BIT ($ *index* $) (*item*)	
39.	CHAR (*floating:item*)	
40.	POS (*file:name*)	
41	NENT (*name*)	
42.	NWDSEN	2.5 3.33

1.	*octal*				2.61	3.33	3.35	3.51
2.	represented by *octal:numerals*							
3.	*octal:constant*			2.63	3.32	4.42	4.43	5.2
4.	O(string of *octal:numerals*)							
5.	*octal:numeral*					2.3	2.61	2.63
6.	∅							
7.	1							
8.	2							
9.	3							
10.	4							
11.	5							
12.	6							
13.	7							
15.	ODD						2.5	3.27
16.	*one:dimensional:constant:list*					4.51	4.55	4.57
17.	BEGIN string of *constants* END							
18.	*one:factor:for:clause*						3.74	3.77
19.	FOR *loop:variable* = *numeric:formula* $							
20.	OPEN					2.5	3.57	3.58
21.	*open:input:statement*						3.5	3.57
22.	OPEN INPUT *file:name* $							
23.	OPEN INPUT *file:name input:operand* $							
24.	*open:output:statement*						3.5	3.58
25.	OPEN OUTPUT *file:name* $							
26.	OPEN OUTPUT *file:name output:operand* $							
27.	*open:statement*							3.58
28.	*open:input:statement*							3.57
29.	*open:output:statement*							3.58
30.	*operand*						3.56	3.58
31.	*input:operand*							3.56
32.	*output:operand*							3.56
33.	there are no other references in this document, but operand also means constant or variable							
34.	*operator*							3.36
35.	*logical:operator* is the only reference in this document							
36.	*optional*	2.62	3.71	4.41	4.56	4.57	4.58	4.59
							5.5	5.6
37.	*optionally*	2.62	2.63	3.71	4.41	4.42	4.51	6.4
38.	OR						2.5	3.36
39.	*or:if:clause*							3.73
40.	ORIF *boolean:formula* $							

1.	*ordinary:entry*				4.55
2.	*entry* of an *ordinary:table*				
3.	*ordinary:entry:description*			4.55	4.56
4.	the set of *item:declarations* and *overlay:declarations* in an *ordinary:table:declaration*				
5.	*ordinary:table*				4.56
6.	*table* specified by an *ordinary:table: :declaration*				
7.	*ordinary:table:declaration*			4.54	4.56
8.	*ordinary:table:item:declaration*			4.55	4.57
9.	ITEM *name* *item:description* $				
10.	ITEM *name* *item:description* $ *one:dimensional:constant:list*				
11.	*ordinary:table:item:name*				4.55
12.	*name* following ITEM in an *ordinary: :table:item:declaration*				
13.	ORIF			2.5	3.73
14.	OUTPUT			2.5	3.58
15.	*output:operand*			3.56	3.58
16.	*array:name*				4.43
17.	*constant*				2.63
18.	*table:name*				
19.	*table:name* ($ *index* $)				
20.	*table:name* ($ *index* ... *index* $)				
21.	*variable*				
22.	*output:parameter*			5.5	5.6
23.	*actual:output:parameter*				
24.	*formal:output:parameter*				
25.	the values and structures for a *procedure:declaration* to produce				
26.	*output:parameter:list*			1.2	3.55
27.	*actual:output:parameter:list*				3.55
28.	*formal:output:parameter:list*				5.5
29.	*output:parameter:statement:name*				5.5
30.	*statement:name* in *output:parameter:list*				
31.	*name* which is followed by *period* in *output:parameter:list*				
32.	*output:statement*		3.5	3.56	3.58
33.	OUTPUT *file:name* *output:operand* $				
34.	OVERLAY	2.5	4.43	4.55	4.56
35.	*overlay:declaration*		4.43	4.55	4.57
36.	*independent:overlay:declaration*				4.43
37.	*subordinate:overlay:declaration*				4.55

1. *packing* 4.59
2. the sharing of computer words by disjunct *items*. This is done only for *table:items* and may be prescribed by *packing:specifications*
3. *packing:specification* 4.56 4.57 4.59
4. D
5. M
6. N
7. parallel 4.56 4.58
8. *table* structure in which there are several blocks, one for each word of an *entry*. The words of a particular *entry* are distributed over these blocks, one per block, and similarly placed in each block
9. *parameter* 3.36 3.55
10. *input:parameter*
11. *output:parameter*
12. *parameter:list* 4.1
13. *input:parameter:list*
14. *output:parameter:list*
15. *parenthesis* 2.4 2.63 3.31 3.33 3.34 3.36 3.71 6.5
16. (
17.)
18. *period* 2.3 2.7 3.33 3.55 3.74 4.1 4.41 5.5
19. .
20. *plus:sign* 2.3
21. +
22. POS 2.5 3.24 3.56
23. precision 2.61
24. number of bits after the binary point
25. *prime* 2.3 2.5 2.8 5.2
26. '
27. *primitive* 2.4 2.5 2.7 2.8 3.1 3.24 3.35 3.36 4.4 4.43 4.51 5.2 5.32 5.6
28. PROC 2.5 5.5 5.6
29. *procedure* 3.55 3.75 4.2 5.4 5.5 6.5
30. subroutine defined by a *procedure:declaration*; sometimes (not in this document) also by a *function:declaration*

1.	*procedure:body*						5.5	5.6
2.	BEGIN *statement:list* END							
3.	*procedure:call:* *:statement*	3.31	3.5	3.55	3.76	4.1	5.5	5.6
								6.5
4.	*procedure:declaration*	3.54	3.55	3.75	3.76	4.1	4.2	4.3
			5.1	5.4	5.5	5.6	6.4	6.5
5.	*procedure:heading*							5.5
6.	*procedure:name*					3.55	5.5	6.5
7.	*name* following PROC in a *procedure:declaration*							
8.	*processing:declaration*	2.1	3.53	3.54	3.76	5.1	5.2	5.3
							5.4	5.5
9.	*close:declaration*							5.4
10.	*function:declaration*							5.6
11.	*procedure:declaration*							5.5
12.	*program:declaration*							5.2
13.	*switch:declaration*							5.3
14.	'PROGRAM						2.5	5.2
15.	*program*	1.3	1.4	2.1	2.4	2.5	2.7	2.8
		3.1	3.33	3.53	3.54	3.71	3.73	3.75
		4.1	4.2	4.4	4.58	5.1	5.2	5.4
					6.1	6.2	6.4	6.5
16.	START *statement:list* TERM $							
17.	START *statement:list* TERM *statement:name* $							
18.	*program:declaration*				4.2	5.1	5.2	5.5
19.	'PROGRAM *name* $							
20.	'PROGRAM *name* *number* $							
21.	'PROGRAM *name* *octal:constant* $							
22.	*program:name*	3.33	3.53	3.75	5.3	5.31	5.32	6.5
23.	*name* following 'PROGRAM in a *program:declaration*							
24.	*record*		3.1	3.24	3.56	3.57	3.58	4.6
25.	the unit of data in a *file* for input or output at one time							
26.	recursive						5.4	5.5
27.	with respect to subroutines, one which calls itself, either directly or indirectly by calling other subroutines which call it in turn							

1.	recursive definition				3.22
2.	definition in which an element of the definition is the term to be defined, perhaps indirectly through a chain of two or more definitions. To be meaningful a recursive definition must incorporate alternative definitions, at least one of which is not recursive. The recursive element then defines structures of arbitrary length				
3.	*relation:list*			3.35	3.36
4.	*dual:relation:list*				3.35
5.	*literal:relation:list*				3.35
6.	*numeric:relation:list*				3.35
7.	*relational*				3.35
8.	pertaining to relationships of equality or ordering between *formulas*				
9.	*relational:operator*		2.4	3.35	3.36
10.	EQ				
11.	GQ				
12.	GR				
13.	LQ				
14.	LS				
15.	NQ				
16.	RETURN			2.5	3.54
17.	*return:statement*		3.5	3.54	5.5
18.	RETURN $				
19.	*right:parenthesis*		2.3	2.4	2.63
20.	)				
21.	*scale*		2.62	2.63	4.53
22.	*number*				2.62
23.	scope				6.5
24.	*sequence*			4.43	4.55
25.	*independent:data:sequence*				4.43
26.	*subordinate:data:sequence*				4.55
27.	parts of *overlay:declarations*				
28.	*sequence:designator*	3.53	5.3	5.31	5.32
29.	*close:name*				5.4
30.	*program:name*				
31.	*statement:name*				3.4
32.	*switch:name*			5.31	5.32
33.	*switch:name* ($ *index* $)				

1.	serial (*table* structure)						*4.56*	4.58
2.	*table* structure in which there is a block for each *entry*, all the words of an *entry* being in the same block							
3.	SHUT					2.5	3.57	3.58
4.	*shut:input:statement*						3.5	*3.57*
5.	SHUT INPUT *file:name* $							
6.	SHUT INPUT *file:name input:operand* $							
7.	*shut:output:statement*						3.5	*3.58*
8.	SHUT OUTPUT *file:name* $							
9.	SHUT OUTPUT *file:name output:operand* $							
10.	*shut:statement*							3.58
11.	*shut:input:statement*							*3.57*
12.	*shut:output:statement*							*3.58*
13.	*sign*	1.3	2.1	2.2	*2.3*	2.4	2.5	2.61
			2.63	2.8	3.26	3.51	3.71	3.75
14.	*letter*							*2.3*
15.	*mark*							*2.3*
16.	*numeral*							*2.3*
17.	*signed*	2.2	*2.61*	2.62	2.63	3.71	4.41	4.42
							4.51	6.4
18.	being preceded by + or - without any intervening *spaces*							
19.	signicand							*2.61*
20.	the significant digits in floating representations of numbers							
21.	*simple*							3.6
22.	in the reference, the property of a *statement* of being a *simple:statement*							
23.	*simple:item*		3.33	4.3	4.42	4.43	4.5	4.53
24.	data structure specified by *simple:item:declaration*							
25.	*simple:item:declaration*			4.4	*4.42*	4.55	5.6	6.5
26.	ITEM *name item:description* $							
27.	ITEM *name item:description* P *constant* $							
28.	ITEM *name constant* $							
29.	*simple:item:name*				4.3	4.43	5.5	6.4
30.	*name* following ITEM in a *simple:item:declaration*							
31.	*simple:statement*	2.1	3.4	*3.5*	3.54	3.71	3.72	3.73
32.	*simple:variable*				3.2	*3.21*	3.23	3.24
33.	*simple:item*							
34.	single letter subscript							2.5
35.	*loop:variable*						*2.5*	*3.74*

1.	*size:specification*	4.56 4.58
2.	*table:size:specification*	4.56
3.	R *number*	
4.	V *number*	
5.	*slash*	2.3
6.	/	
6.	*space*	2.2 2.3 2.4 2.5 2.63 2.7 2.8 3.71 4.41
7.	the *mark* represented with no ink on the paper	
8.	*special:compound*	3.74 3.75 3.77
9.	BEGIN *statement:list if:clause* END	
10.	*specification*	4.59
11.	*basic:structure:specification*	4.56
12.	*packing:specification*	4.56
13.	*table:size:specification*	4.56
14.	*star*	2.3
15.	*	
16.	START	2.5 6.2
17.	*statement*	2.1 3.1 3.4 3.51 3.53 3.54 3.55 3.57 3.58 3.6 3.72 3.73 3.74 3.75 4.1 4.3 4.56 4.57 5.1 5.2 5.3 5.31 5.32 5.4 5.5 5.6 6.2 6.4 6.5
18.	*complex:statement*	3.7
19.	*alternative:statement*	3.73
20.	*conditional:statement*	3.72
21.	*direct:statement*	3.71
22.	*loop:statement*	3.74
23.	*independent:statement*	3.4
24.	*compound:statement*	3.6
25.	BEGIN *statement:list* END	
26.	*simple:statement*	3.5
27.	*named:statement*	3.4
28.	*name* . *statement*	
29.	*statement:list*	3.6 3.74 3.75 5.4 5.5 6.2
30.	*statement*	3.4
31.	*declaration statement:list*	
32.	*directive statement:list*	
33.	*statement:list declaration*	
34.	*statement:list directive*	
35.	*statement:list statement*	

1. *statement:name* 3.33 3.4 3.53 3.54 3.55 3.72 3.73 3.74 3.75 4.1 4.6 5.3 5.5 6.2 6.5
2. *status* 2.61 3.2 3.23
3. *status:assignment:statement* 3.51
4. *status:variable* = *status:formula* $
5. *status:constant* 2.63 2.8 3.32 3.51 3.55 4.41 4.6 5.32 6.5
6. V (*letter*)
7. V (*name*)
8. *status:formula* 3.32 3.35 3.36 3.51
9. *status:constant* 2.63
10. *status:function*
11. *status:variable*
12. *status:function* 3.32
13. invocation of a *function:declaration* with a *status* output value
14. *status:item* 4.42 6.5
15. *item* specified by *declaration* in which the *item:name* is followed by S
16. *status:item:description* 4.41
17. S string of *status:constants*
18. S *number* string of *status:constants*
19. *status:item:name* 2.63 4.6
20. *name* following ARRAY or ITEM or STRING and followed by S in an *item:declaration*
21. *status:list* 4.6
22. string of *status:constants*
23. *status:variable* 2.61 3.32 3.36 3.51 3.52 3.55
24. *status:item*
25. STOP 2.5 3.4 3.54 5.3
26. *stop:statement* 3.4 3.5 3.54 6.1
27. STOP $
28. STOP *statement:name* $
29. STRING 2.5 4.57

1. string 3.58 3.6 3.71 3.74

2. in reference to some sort of element, one such element or an arrangement of more than one with one element following another

3. in strings of *signs* used to form *symbols*, there is, in general, no separation between the *signs*

4. in strings of *symbols*, they are separated by *spaces* or *comments*

5. *string* 3.1 4.5 4.57

6. collection of data declared by a *string:item:declaration*

7. essentially equivalent to *string:item* but *string* has the connotation of the group of data

8. *string:item* 3.31 4.5 4.57 4.58 5.31

9. collection of data declared by a *string:item:declaration*

10. essentially equivalent to *string*, but *string:item* has the connotation of a member of the group

11. *string:item:declaration* 4.57 4.58

12. *string:item:name* 4.5

13. *name* following STRING in a *string:item:declaration*

14. *subordinate:data:sequence* 4.55

15. part of an *ordinary:entry:description*

16. *subordinate:overlay:declaration* 4.55

17. part of an *ordinary:entry:description*

18. *subordinate:overlay:specification* 4.55

19. part of an *ordinary:entry:description*

20. subroutine 5.4 5.5

21. a piece of programming which can be utilized at various points in a program. In a JOVIAL *program* subroutines can be set up by means of *close:declarations*, *function:declarations*, and *procedure:declarations*

1.	subscript							
2.	*index*							3.22
3.	*loop:variable*							2.5
4.	because of ambiguity it is recommended that this term not be used unless the meaning is clear from context. This is usually not the case since *loop:variables* are often used as *indices*							
5.	SWITCH					2.5	5.31	5.32
6.	*switch*		3.75	5.3	5.31	5.32	5.4	5.5
7.	*index:switch*							
8.	*item:switch*							
9.	*switch:declaration*	3.53	3.75	3.76	4.6	5.1	5.3	5.32
								5.4
10.	*index:switch:declaration*							5.31
11.	*item:switch:declaration*							5.32
12.	*switch:list*						5.31	5.32
13.	*index:switch:list*							5.31
14.	*item:switch:list*							5.32
15.	*switch:name*			3.75	5.3	5.31	5.32	6.5
16.	*name* following SWITCH in a *switch:declaration*							
17.	*symbol*	2.1	2.2	2.4	2.5	2.62	2.63	2.7
					2.8	3.71	3.75	6.3
18.	*abbreviation*							2.5
19.	*comment*							2.5
20.	*constant*							2.63
21.	*ideogram*							2.5
22.	*loop:variable*						2.5	3.74
23.	*name*							2.5
24.	*primitive*							2.5

1.	TABLE				2.5	4.56	4.58	4.59
2.	*table*	3.1	3.22	3.24	3.28	3.33	3.55	3.56
		3.74	4.2	4.43	4.5	4.51	4.54	4.55
					4.56	4.57	4.58	4.59
3.	data structure, a collection of *items* organized by a *table:declaration*							
4.	*table:declaration*		3.21	4.4	4.54	4.55	4.56	4.57
5.	*defined:entry:table:declaration*							4.58
6.	*like:table:declaration*							4.59
7.	*ordinary:table:declaration*							4.56
8.	*table:entry*					3.56	4.5	4.55
9.	the set of all the *items* of a *table* with the same second component of the *index* in the case of *strings* and the same *index* in the case of other *items*							
10.	*table:item*						3.33	4.58
11.	*item* specified by a *table:item:declaration*							
12.	*table:item:declaration*						4.4	4.55
13.	*defined:entry:item:declaration*							4.57
14.	*ordinary:table:item:declaration*							4.55
15.	*string:item:declaration*							4.57
16.	*table:item:name*							4.43
17.	*name* following ITEM or STRING in a *table:item:declaration*							
18.	*table:name*	3.33	3.55	3.56	3.74	4.43	4.56	5.5
								6.5
19.	the *name*, if there is one, immediately following TABLE in a *table:declaration*							
20.	*table:size*							4.59
21.	number of *entries* in a *table*							
22.	*table:size:specification*					4.56	4.58	4.59
23.	R *number*							
24.	V *number*							
25.	TERM					2.5	6.2	6.4
26.	TEST					2.5	3.54	3.77
27.	*test:statement*					3.5	3.54	3.77
28.	TEST $							
29.	TEST *loop:variable* $							
30.	*three:dimensional:constant:list*							4.51
31.	BEGIN string of *two:dimensional:constant::lists* END							
32.	*transmission:code*	2.61	2.63	3.2	3.26	3.35	3.51	4.6
33.	pertaining to the computer-independent encoding of *signs* which is a standard for JOVIAL							

1.	*transmission:code:constant*	2.2 2.63 3.32
2.	*number*T(*signs*)	
3.	*number* is the number of *signs*	
4.	*transmission:code:formula*	3.32
5.	*octal:constant*	2.63
6.	*transmission:code:constant*	2.63
7.	*transmission:code:function*	
8.	*transmission:code:variable*	
9.	*transmission:code:function*	3.32
10.	invocation of a *function:declaration* with a *transmission:code* output value	
11.	*transmission:code:item:description*	4.41
12.	T *number*	
13.	*transmission:code:variable*	3.2 3.32 4.6
14.	*transmission:code:item*	
15.	*item* specified by *item:declaration* in which T follows the *item:name*	
16.	BYTE ($ *index* $) (*transmission:code:item*)	
17.	truncated	3.22
18.	part removed from the left or right	
19.	with *numeric* values, if left or right is not stated, usually from the right	
20.	with *numeric* values truncated on the right, care will usually be taken to insure that the remaining value will be the same as if the computer representation were "sign on the left followed by magnitude bits"	
21.	*two:dimensional:constant:list*	4.51 4.52 4.57
22.	BEGIN string of *one:dimensional::constant:lists* END	
23.	*two:factor:for:clause*	3.74 3.77
24.	FOR *loop:variable* = *numeric:formula*, *numeric:formula* $	
25.	*variable*	2.6 2.61 3.1 3.2 3.21 3.22 3.23 3.24 3.25 3.26 3.27 3.3 3.33 3.51 3.52 3.55 3.56 3.71 4.41 4.56 4.6
26.	*boolean:variable*	3.27
27.	*dual:item*	
28.	*entry:variable*	3.28
29.	*literal:variable*	3.2 3.26
30.	*hollerith:variable*	
31.	*transmission:code:variable*	

1.	*numeric:variable*	<u>3.2</u>
2.	*fixed:variable*	<u>3.25</u>
3.	*floating:item*	
4.	*integer:variable*	<u>3.24</u>
5.	*status:item*	
6.	*variable:length:table*	3.33
7.	*table* specified by a *table:declaration* in which V follows the *table:name* or the *primitive* TABLE	

APPENDIX

The complete specification of a procedure-oriented programming language seems to be a difficult task. At any rate all attempts, so far, to write specifications for languages as complex as JOVIAL, ALGOL, or COBOL have not been particularly successful. That is not to say such writeups have failed to please anyone. Indeed, some such descriptions have been well received by some workers in the field, but in each case there has been a significant segment of the computing community that has been dissatisfied.

The author of this document is interested in knowing how close he has come to producing an easily understood and complete description of JOVIAL (J3). The following page may be torn out and returned with an indication of the reader's opinions. Lengthier responses, in the form of letters, will be most welcome.

To: Millard H. Perstein
System Development Corporation
2500 Colorado Avenue
Santa Monica, California

Room 2328

The over-all presentation of the language is: Very clear and orderly ______

Clear enough for tutorial purposes _____ Confused or confusing _______

Complete but difficult to grasp _______ Extremely garbled _______

Other remarks:

The use of the special metalanguage is: Very helpful ________

An obstacle to understanding ________ Of some value ________

Other remarks:

In comparison with other metalanguages used in describing programming languages (JOVIAL or others) the present one:

Is a happy blend of brevity and clarity ________

Is too long-winded______ Is too cryptic ________

Other remarks:

The special type face for metalanguage phrases is: Very helpful _____

Not as good as special brackets_____ A strain on the eyes _________

Other remarks:

General remarks and suggestions for improvement:

Changes or corrections which should be issued immediately as modifications to this document:

Name__________________________ Position______________________________

Organization __________________________

To: Millard H. Perstein
System Development Corporation
2500 Colorado Avenue
Santa Monica, California

Room 2328

The over-all presentation of the language is: Very clear and orderly ______

Clear enough for tutorial purposes _____ Confused or confusing _______

Complete but difficult to grasp ________ Extremely garbled _______

Other remarks:

The use of the special metalanguage is: Very helpful _________

An obstacle to understanding _________ Of some value _________

Other remarks:

In comparison with other metalanguages used in describing programming languages (JOVIAL or others) the present one:

Is a happy blend of brevity and clarity _________

Is too long-winded_______ Is too cryptic _________

Other remarks:

The special type face for metalanguage phrases is: Very helpful _____

Not as good as special brackets_____ A strain on the eyes __________

Other remarks:

General remarks and suggestions for improvement:

Changes or corrections which should be issued immediately as modifications to this document:

Name___________________________ Position_______________________________

Organization ___________________________

SDC

SYSTEM DEVELOPMENT CORPORATION / 2500 COLORADO AVENUE / SANTA MONICA, CALIFORNIA