GENERAL ELECTRIC
COMPUTERS

GE-625/635 JOVIAL USER'S INFORMATION

PRELIMINARY

March 1967

GENERAL ELECTRIC

G E - 6 2 5 / 6 3 5 J O V I A L

U S E R ' S I N F O R M A T I O N

March 1967

GENERAL ELECTRIC

INFORMATION SYSTEMS DIVISION

PREFACE

This publication is an interim document which contains information on how to use the GE-625/635 JOVIAL compiler under the control of the GECOS operating system. It provides the information necessary to set up a deck for compilation and subsequently to execute the compiled program. An extended explanation of each error message produced by the compiler is included, although this is needed in only a few cases.

For a concise description of the J3 JOVIAL language implemented on the GE-625/635 computer, the user should refer to JOVIAL Compiler Programming Reference Manual, CPB-1201. Additional helpful information can be found in GE-625/635 General Loader, CPB-1008 and GE-625/635 Programming Reference Manual, CPB-1004. This, when combined with the user information contained here and a working knowledge of the GECOS operating system, should enable an experienced programmer to write and debug JOVIAL programs on the GE-625/635.

Chapter 1 discusses the three phases of the JOVIAL compiler and gives information concerning error detection and debugging. Chapter 2 describes the processes and procedures involved in a JOVIAL compilation. The appendixes provide supplemental information.

Suggestions and criticisms relative to form, content, purpose, or use of this manual are invited. Comments may be sent on the Document Review Sheet in the back of this manual or may be addressed directly to Engineering Publications Standards, B-90, Computer Equipment Department, General Electric Company, 13430 North Black Canyon Highway, Phoenix, Arizona 85029.

CONTENTS

ILLUSTRATIONS

I. THE JOVIAL SYSTEM

THE COMPILER

The GE-625/635 JOVIAL compiler comprises three programs, which are called phases. Each phase operates on the entire JOVIAL program to be compiled and then passes its accrued information on to the next phase, either by leaving this accumulated data in common storage, if space requirements permit, or by leaving portions in common storage and using I/O devices for the remainder.

Phase One

In phase one, each complete JOVIAL statement is processed through that phase before the next statement is examined. Phase one converts the JOVIAL statement to a form called Polish notation, which is derived from the syntax or grammatical rules of the language. Each statement remains in this type of notation through the rest of the compilation process. It is this Polish notation which can be temporarily stored on I/O devices between phases. The data descriptions and actual preset data remain in store* from phase to phase.

Phase Two

Phase two takes each statement, now in Polish, and processes it along with contiguous statements that constitute a logical set. This phase processes by blocks of code rather than by single statements. This allows certain optimization of code across statements, which the programmer may recognize upon examining the machine code produced. The information necessary to produce binary code is passed from phase two to phase three. Again, if necessary, temporary I/O storage may be utilized.

*In conformity with the IFIP/ICC vocabulary, the device formerly called "memory" is now called "store."

Phase Three

Phase three, in essence, is the assembly portion of the compiler. It converts the output from phase two into machine code instructions for printing and into a binary card image file for immediate or later execution or both.

In accordance with system requirements, the compiler will accept one program for compilation for each $JOVIAL card input to the GECOS system. The input is in the form of JOVIAL statements on cards and the output is an optional binary deck for later execution, and an optional binary file for immediate execution, and optional listings are produced for the source input and the machine code output. This information is described in detail in subsequent sections.

ERROR DETECTION

All phases of compilation have error detection. Syntactical errors and semantic errors are noted by error messages. The compiler error detection is related to the programmer by terse statements. If an error is found during phase one, the error message follows the JOVIAL statement. Errors found during phase two cause error messages to be grouped at the end of the source listing. All error messages have JOVIAL statement numbers associated with them, and those detected in phase one have an arrow pointing to the column of input data that the compiler interprets as the cause of the error.

Also associated with the error message in phase one is a number giving the approximate character position within the statement where the error was detected. This number is used to determine the line position within a multicard statement. The arrow and character count may not always be accurate because of the tree or branch method of determining the syntax of a statement, but when considered with the error message, they are sufficient to locate the error. These error messages are described in Appendix A.

OBJECT CODE DEBUG

By specifying the proper option on the $JOVIAL card, a programmer may have the compiler build DEBUG SYMBOL TABLE cards. This information is, in turn, used by the system loader to provide debug information at execution time. This option is discussed fully in Appendix B.

2. JOVIAL COMPILATION

INPUT

Control Information

The compiler control parameters are input by the GECOS system control card used to initiate a JOVIAL compilation. Any of the following parameters may be present or absent from the control card. Parameters must start in column 16, be separated by commas, and may appear in any order. When a parameter is omitted, the predetermined value of the pair of values is assumed. The underlined parameter for each pair in the following list is the one assumed.

DECK	Produce a binary deck for later execution
NDECK	Do not produce the deck
LSTIN	Produce a listing of the source program
NLSTIN	Do not produce the listing
LSTOU	Produce a binary symbolic (GMAP) listing of the compiled program
NLSTOU	Do not produce the listing
COMDK	Produce a compressed source deck from the input deck
NCOMDK	Do not produce the deck
DEBUG	Produce a DEBUG SYMBOL TABLE for optional use by the system loader at execution time
NDEBUG	Do not produce the table

The following is an example of a JOVIAL control card. The assumed values in this case are DECK, LSTIN, and NDEBUG.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
$							J	O	V	I	A	L			L	S	T	O	U	,	C	O	M	D	K									

Source Language

Card Format. The JOVIAL source card has a free-field format from column 1 to 72. Columns 73 to 80 are reserved for deck identification and sequencing and are listed by the compiler. Column 72 of one card and column 1 of the next card are considered to be contiguous. No source card may have a $ in column 1. Direct code formats are shown in Appendix D.

Program Format. The first statement in a JOVIAL program must be a START. The last statement must be a TERM$.

Deck Identification

Immediately preceding the START card in the user's deck, there must be an identification card. The compiler uses this card to create a SYMDEF to be used by GELOAD. There are three types of source input to the compiler: a program, a procedure, and data to generate a COMPOOL. (See Appendix C.) The format of the identification card for each of these is described below.

Program

Columns 1 to 6 contain PROGRM. Columns 8 to 13 may be blank or may contain a name of six characters or less, left-justified, to be used as the SYMDEF. If no name is present, a SYMDEF of six periods is created.

Procedure

Columns 1 to 6 contain PROCED. No other identification is necessary. The name of the procedure is used as the SYMDEF.

COMPOOL Generation

Columns 1 to 6 contain GENCOM.

OUTPUT

Listings

There are three kinds of printer listings:

Source Language Listing. The source language listing consists of one-line-per-card replication of the JOVIAL input deck. (See Figure 1.)

Each line of source output has a number at the left-hand margin. This is the statement number of the last statement begun on that line. Statement numbering begins with zero for the START card. There also is a number at the right margin of the listing. This is a one for one input card count for altering purposes.

Object Code Listing. There is an object code listing that corresponds to the output of the compiler. (See Figure 2.) The object code listing resembles an assembly program listing, and the following fields are located from left to right on the page.

(1) Store location. This is relative to a base address of 0 since this is a relocatable binary deck.

(2) Octal representation of binary (12 digits).

(3) Octal representation of relocatable bits.

```
0238      SR10.              IF HH $                                     0203
0240                           BEGIN       TTCAP = TTCAP+TCAP $          0204
0242                COMP = CC($B$) $          YES($B$) = 1 $             0205
0246                          PRNT  $   CAP = 0 $ TCAP = 0 $ HH = 0 $    0206
0247                          GOTO  SR2 $                                0207
0248                          END                                        0208
```

Figure 1. Source Language Listing

(4) Label field. This may or may not exist. If there is a label, it will be the original JOVIAL label or a generated label of the form nG where n is an integer.

(5) Machine operation mnemonic.

(6) Operand(s).

(7) Source statement line number.

Error Messages. Compiler error messages are of two general types, syntactical (construction) errors and semantic (usage) errors. The syntactical errors are detected during the first phase of compilation. Semantic errors are found during subsequent phases of compilation. Appendix A contains a description of JOVIAL syntactical and semantic errors. As previously described, syntactical errors detected in phase one are noted by error messages that will be printed below the incorrect statement. Errors that cannot be detected at the exact time of occurrence (for example, BEGIN and END out of phase) will be noted when they become apparent. Even when critical errors occur in a program being compiled, the compiler will attempt to continue compilation.

Binary Output

If execution is specified, the compiler produces a binary card image file for execution immediately after compilation. If a binary deck is requested by the $JOVIAL control card, binary cards are produced. In either case, the deck structure produced is identical to that produced by a GMAP assembly described in the GE-625/635 General Loader manual under the heading "Relocatable Deck Description."

```
*   1          2            3     4          5           6          7

    000241  002671  4500  00  010    8G         STZ         A          000110
    000242  002671  7220  00  010    11G        LDLX2       A          000110
    000243                           I005       BSS         0          000112
    000243  000000  2350  03  000    A4         LDA         0,DU       000112
    000244  102045  0750  03  010               ADA         INP+1,DU   000112
```

* The seven fields are labeled only to make this exhibit more meaningful.

Figure 2. Object Code Listing

APPENDIX A

JOVIAL COMPILATION ERROR MESSAGES

If, because of an error, a JOVIAL statement is not compiled, binary code to call the GECOS system is inserted and flagged with an E in the binary listing. (See Figure 3.)

Syntactical errors are detected in phase one and are noted by error messages which are printed below the JOVIAL statement in error. (Refer to Figure 4.) In the example there is no END for the TABLE declaration. The compiler discovers this upon encountering the symbol DEFINE in statement 6. The arrow indicates the point of discovery, as does the error character count 000000. In this particular occurrence there is no effect on the compilation, since the compiler assumes that the TABLE declaration is finished, which it is.

The following list contains all current JOVIAL compilation error messages and their meanings and action.

Statement	Meaning and Action
ALLOCATED MEMORY IS EXHAUSTED	The compiler needs more storage and it is not available. Compilation is terminated.
ALLOCATION TROUBLE	The data being compiled are allocated inconsistently in an OVERLAY statement. Erroneous data allocation may result.

```
0211          BEGIN WS9 = WS9 + RANGE $                          0196
0212                      WS10 = WS10 + 1 $                      0197
0213                WS11 = (RANGE*10)/RAD-.4999 $                0198
0214           FREA($WS11$)=FREA($WS11$)+1 $                     0199
```

Source Listing Excerpt

```
**ERR**STA    0086       TOO FEW SUBSCRIPTS
**ERR**STA    0089       TOO FEW SUBSCRIPTS
**ERR**STA    0158       PRECLUDED BY CONTEXT
**ERR**STA    0214       NOT TABLE ITEM
**ERR**STA    0214       NOT TABLE ITEM
**ERR**STA    0283       NOT TABLE ITEM
```

Semantic Errors Found in Phase Two

```
   001255  216000  4350  03  000          UFA        216000,DU      000213
   001256  021367  7560  00  010          STQ        WS11           000213
 E 001257  206502  2360  00  000          LDQ        206502         000214
 E 001260  000010  0010  00  000          MME        10             000214
   001261  006400  4310  03  000          FLD        6400,DU        000215
   001262  021350  5650  00  010          FDV        RANGE          000215
```

Machine Code Generated

Figure 3. Usage Error Example

```
 0002         TABLE TABA V 5 P $                                   GE080220     0021
 0003           BEGIN ITEM TAA A 10 S 5 $                          GE080230     0022
 0003             BEGIN 1.0A5 2.0A5 3.00A5 4.0A5 END               GE080240     0023
 0004                 ITEM TAB A 20 S 5 $                          GE080250     0024
 0004            BEGIN 1084.80A5 1084.89A5 END                     GE080260     0025
 0005                 ITEM TAC A 10 S 5 $                          GE080270     0026
 0005             BEGIN -8.0A5 END                                 GE080280     0027
 0006         DEFINE IFEITHER "IFEITH" $                           GE080290     0028
                   ↑                                                     000000
**ERR**STA    0006     MISSING END
```

Figure 4. Syntax Error Example

Statement	Meaning and Action
ALREADY ACTIVE	The FOR statement being compiled is using a loop variable that has already been activated by a prior FOR. The fact is ignored, but it may result in erroneous code.
ARRAY EXCEEDS REASONABLE SIZE	The space required for ARRAY is greater than $24{,}000_{10}$. No space is allocated.
BAD STATUS RELATIONSHIP	The status constant being compared to this status variable is not one of the previously declared legal values. The statement is not compiled.
BAD XEC NAME	The XEC parameter is not a procedure call or a zero. The statement is not compiled.
CIRCULAR DEF	The expression being compiled contains a redefinition (by the DEFINE declaration) which is improperly constructed. The statement using this definition is not compiled.

Statement	Meaning and Action
COMPILER ABORT. SEND DUMP TO GENERAL ELECTRIC	There is a logic error in the compiler.
COMPOOL NOT MOUNTED	A COMPOOL list has been encountered, but no COMPOOL input tape was mounted. COMPOOL list is not compiled.
CONFLICTING USE	The variable being compiled is being used in an improper context. The statement is not compiled.
CONST OVFLO	The exponent part of the constant being compiled is larger than 10^{28}. The exponent is set to zero.
EXIT BUFFER HAS OVERFLOWED	The compiler has used up the allocated buffer area for its recursive calls. Compilation is terminated.
FIELD TOO BIG	An item has been declared too large for its type. The declaration is not compiled.

Statement	Meaning and Action
FILE NAME REQUIRED	An I/O request has been found without a file name in the parameter reserved for it. The I/O statement is not compiled.
ILLEGAL CHARACTER	A character input from the source deck is illegal. It is ignored.
ILLEGAL CONVERSION	In the assignment or exchange statement being compiled, the data types are incompatible; that is, no implied conversion is possible. The statement is not compiled.
ILLEGAL DATA LENGTH PARAMETER	The parameter defining the length of the data in the I/O transfer is illegal. The statement is not compiled.
ILLEGAL FOR COMPOUND	An IF statement followed immediately by an END, which is not associated with a FOR, is an illegal construction. The IF statement is not compiled.

Statement	Meaning and Action
ILLEGAL PRIMITIVE	A name beginning with a prime character has been encountered, but it is not among the allowable primitives. The statement is not compiled.
ILLEGAL STATUS ASSIGNMENT	The status constant being assigned to this status variable is not one of the previously defined legal values. The statement is not compiled.
ILLEGAL TEST	The TEST statement is attempting to test a loop variable that is not active. The TEST statement is not compiled.
INCONSISTENTLY DECLARED	The data being compiled are improperly declared. The declaration is not compiled.
JUNK PROGRAM	A combination of errors so extensive that the compilation cannot continue has been detected; compilation terminates
LOOP CONTROL	Multiple three-factor FOR statements are used in parallel. Erroneous code may be generated.

Statement	Meaning and Action
MISSING ASTERISK	The asterisk (*) is omitted from the exponentiation brackets. It is assumed.
MISSING BEGIN	There is an absence of a BEGIN in a PROC declaration. The BEGIN is assumed.
MISSING DOLLAR	The dollar sign ($) is missing from a subscript bracket or a statement terminator. It is assumed.
MISSING END	In processing a statement in which an END is required, the compiler has encountered a symbol that is not syntactically correct and that is not an END. Or, the end of a program was reached with a count of begins greater than the count of ends. The effect on compilation depends on the particular occurrence.
MISSING SLASH	The slash (/) is omitted from the absolute value brackets. It is assumed.
MISSING START	The START card does not appear first in the JOVIAL program deck. It is assumed

Statement	Meaning and Action
MUL DEF	The statement name currently being used has not been defined more than once in this program. Erroneous references to this label result.
MULTI STA ERROR	This is a combination of errors in contiguous statements. The statements are not compiled.
MULTIPLE UNARIES	Multiple arithmetic or Boolean unary operators (-, +, NOT) occur in an expression. The expression is evaluated as it stands.
NAME NOT IN COMPOOL	A COMPOOL list contains a name that is not in COMPOOL. The name is ignored.
NESTED PROC	The PROC statement being compiled is in the scope of another procedure declaration. The PROC statement is not compiled.
NO PATTERN TABLE	No pattern table was declared for this table, which was declared as LIKE. The LIKE declaration is not compiled.

Statement	Meaning and Action
NO PROC FOR RETURN	A RETURN statement appears outside the scope of a procedure declaration. The RETURN statement is not compiled.
NOT TABLE ITEM	A name is subscripted that is not declared in a table. The statement is not compiled.
PRECLUDED BY CONTEXT	A comparison of the usage of this variable or constant and its type indicates an inconsistency. The statement is not compiled.
PRESET CONST ERROR	Something other than a legal constant is in PRESET LIST or there is a missing END. Remaining preset constants are invalid.
SUPERFLUOUS END	There is an extra END or missing BEGIN in the program. This can be an indication of possible serious trouble. The effect on the compilation depends on the particular occurrence.

Statement	Meaning and Action
SYNTAX	An error is detected in the construction of the statement currently being compiled. The statement is not compiled.
TWO FEW ARGS	The BIT or BYTE modifier being compiled does not have any defining parameters. The statement is not compiled.
TOO FEW SUBSCRIPTS	The subscripted item being compiled has fewer subscripts than were contained in its original definition. The statement is not compiled.
TOO MANY ARGS	The BIT or BYTE modifier being compiled has more than two defining parameters. The statement is not compiled.
TOO MANY SUBSCRPTS	The subscripted item being compiled has more subscripts than were contained in its original definition. The statement is not compiled.

Statement	Meaning and Action
UNDEF	The statement name currently being used has not been defined in this program. Erroneous transfers may result.
WORK BUFF HAS OVERFLOWED	The compiler has used up the allocated working area. Compilation is terminated.
$ IN COMMENT	A $ sign was encountered while processing a comment. Everything up to the $ is considered a comment, and the symbol after the $ is considered the beginning of a new statement.

APPENDIX B

OBJECT CODE DEBUG OPTION

The JOVIAL compiler generates a complete Debug Symbol Table when the debug option is specified on the $JOVIAL card. The user then may reference this information at load time. If he chooses to do so. LOADER DEBUG control cards must be included in his deck. These control cards are specified in the GE 625/635 General Loader manual.

There is not a one-to-one correspondence between the types of data used in the JOVIAL language and the types of data expected by the loader. The following lists give the data types that appear on the .SYMT. cards for JOVIAL declared data.

JOVIAL	.SYMT.
Boolean	Logical
Integer Double	Double Precision
Floating	Real
Integer Single	Integer
Status	Integer
Fixed Single	Octal
Fixed Double	Octal
Hollerith	Octal
Transmission	Octal
Statement Labels	Instruction
Procedure Names	Instruction

The loader does not accept symbols in excess of six characters. The compiler uses the first six characters of a name to meet this requirement.

APPENDIX C

COMPOOL

INTRODUCTION

The COMPOOL concept used for the JOVIAL compiler for the GE-635 is that of an auxiliary data declaration source. This appendix describes its preparation and use. The COMPOOL consists of JOVIAL declarations adhering to all the rules that apply to those declarations in the language. The COMPOOL can be used to control the environment of a system of programs and it also can serve as a basis for test design by using the preset feature to initialize inputs.

COMPOOL PREPARATION

Legality Checking

The data to be used to construct the COMPOOL are first passed through phase one of the compiler as a JOVIAL program in order to error-check it. All the rules which apply to other data declarations of a JOVIAL program apply to COMPOOL data. The only difference is that the identification card, which always precedes the START statement, contains GENCOM in columns 1 to 6. If no errors are detected, the COMPOOL generated program is called into store to create the COMPOOL file. If there are errors, the run is terminated at the end of phase one and a source listing is made, showing any detected errors. The same error messages applicable to a JOVIAL program's declarations are valid for COMPOOL data. The deck set up is shown in Figure 5.

COMPOOL Generation

After normal error checking by phase one is complete, the COMPOOL generator is loaded. This program operates on the JOVIAL input cards as they are given to the compiler. It creates a directory of names of the data in the COMPOOL and the number of card image records of each unique set of data. A TABLE includes all the items declared in that table. This directory is a record that precedes the actual data on the COMPOOL file. The COMPOOL file then consists of a directory of data names, each of which must be 12 BCD characters or less, followed by the card images of the JOVIAL data.

COMPOOL USE

To use the COMPOOL declared data, a programmer uses a COMPOOL statement followed by the list of names from the COMPOOL that he wishes compiled into his program. These names are bracketed with BEGIN and END. This COMPOOL statement must appear in the JOVIAL program prior to the use of the data in a JOVIAL statement or a declaration requiring a pattern table from the COMPOOL. Figure 6 gives a sample portion of a program using this feature.

In this example, all of the items in table ALPHA are compiled into the user's program. Table items are never requested by name; only tables, simple items, and arrays are so requested.

The programmer must request to the operating system that the COMPOOL file be mounted prior to a compilation requiring data from that COMPOOL.

```
$SNUMB
$IDENT
$JOVIAL

$LIMITS
$DATA     S*
$INCODE IBMF

GENCOM
START



DATA DECLARATIONS

TERM$
$END JOB
```

Figure 5. COMPOOL Generation Deck Setup

```
START "PROGRAM A"
 ITEM HOL 120 H $
 COMPOOL
  BEGIN
   ALPHA   BETA   AA
   ARRAYONE   BOOL
  END
 TABLE ALPHA1 L $
 IF BOOL $
  BEGIN
   TEST1$
   TEST2$
  END
```

Figure 6. COMPOOL Use

APPENDIX D

DIRECT CODE

INTRODUCTION

The DIRECT statement in the JOVIAL language serves to define a set of operations which must be expressed in machine-oriented language. This language consists of legal GMAP codes. Each line of machine code is considered a symbol; thus, ASSIGN statements and DIRECT or JOVIAL brackets must not appear on lines containing machine code instructions. Direct code may address locations designated by names defined in JOVIAL. A label of a direct code instruction becomes a defined statement name as though defined on a JOVIAL statement. A statement name prefixed to the DIRECT bracket is considered to designate the first executable instruction within the DIRECT JOVIAL brackets. Index registers used within DIRECT JOVIAL brackets should be saved and restored within those brackets.

INSTRUCTION REPERTOIRE

The legal machine codes for use in DIRECT code are operation codes listed in Appendix B of the GE 625/635 Programming Reference Manual with the following inclusions: SXLn and LXLn. In addition, the following two directives (pseudo codes) are allowed: ZERO and SYMREF.

INSTRUCTION FORMAT

The machine code instructions within the DIRECT code must adhere to the following format:

Columns 1-6	label field

The label must be a legal GMAP label (that is, at least one alpha) containing no characters other than periods, alphabetic characters, or numerals.

Column 7 even-odd field

The letter E causes the instruction to be given an even address, and the letter O causes the instruction to be given an odd address. A blank causes no significance to be given to addressing in terms of even or odd. Any other character is considered illegal.

Columns 8-15 operation field

Instruction mnemonic begins in column 8.

Columns 16-72 location and comment field

This field has three subfields: subfield 1, subfield 2, and subfield 3.

- o Subfield 1 may contain an operand.
- o Subfield 2 may contain an operand or a modifier.
- o Subfield 3 may contain comments.
- o Subfield 1 and subfield 2 are separated by a comma, and subfield 2 and subfield 3 are separated by a blank.

An operand may be a JOVIAL name or label or a legal DIRECT machine code label or a constant of the form d, where d is a pure numeric decimal, or a constant of the form = 0 dd, where dd is an octal number. Modifier is a legal GMAP modifier (such as DU). Any blank encountered in columns 16-72 terminates subfield 2 and causes the remaining columns to be considered as comments.

Decimal increments or decrements are allowed in subfield 1 and subfield 2. Either subfield may contain an asterisk which is translated as the address of the instruction of which this operand field is a part.

Special consideration is given to the instructions STCA and STCQ: subfield 2 must contain two octal characters whose bit configuration represent the character positions of the data referenced in subfield 1 which will be stored.

APPENDIX E

RESTRICTIONS

The following are language variations in GE-625/635 JOVIAL:

DUAL items are excluded

STRING items are excluded

Medium table packing is treated like dense table packing

The J3X I/O are implemented

GE-625/635 GMAP assembly code (excluding all MACRO instructions) is the only legal type of direct code

To conform with the conventions of GECOS, a $ must not appear in column 1 of an input card

To conform with the conventions of GELOAD, procedure names for which SYMREFS must be generated must not exceed six characters.

APPENDIX F

FILE STATUSES AND I/O FUNCTION CODES

The GE-625/635 JOVIAL compiler will implement the J3X I/O as defined in JOVIAL Programming Reference Manual. Although file declarations, IN statements, OUT statements and IOn statements are completely described in that manual, file statuses and function codes for file manipulation and data transmission are not listed; therefore, they are included here.

STATUSES

File status values which can be returned to an operating program are:

Numeric Value	Interpretation
0	Normal termination after I/O
1	Null
2	Segment Mark (Reading only - a one-character record other than octal 17 was encountered)
3	End-of-File (A one-character record of octal 17 was encountered, or file has been closed by an OUT statement)
4	Buffer-length error (Input area smaller than logical record)
5	Parity error

FUNCTION CODES

IN Statements

For IN statements of the form IN (FCN, filename) $, FCN may be any one of the following values:

Numeric Value	Interpretation
1	Open Input file
2	Close Input file
4	Close Input with rewind
5	Release Input reel
6	Close Input and open output

OUT Statements

For OUT statements of the form, OUT (FCN, filename) $, FCN may be:

Numeric Value	Interpretation
1	Open Output file
2	Close Output file
3	Close Output with rewind
4	Force Output End-of-Reel

Date Transmission Requests

For data transmission request of the form IOn (FCN, filename, XEC, beginning address, length) $, FCN may be:

Numeric Value	Interpretation
0	Read logical record

Numerical Value	Interpretation
1	Read segment (up to next one-character record not octal 17--Transmit data to store).
3	Transmit print image
4	Write logical record
5	Punch binary record
6	Punch BCD record
7	Write segment mark (one-character record, not octal 17, 75, 76, or 77)

Device Manipulation Request

For a device manipulation request of the form, IOn (FCN, filename, XEC) $, FCN may be:

Numeric Value	Interpretation
8	Move forward one logical record
9	Move forward past segment mark (or end-of-file mark)
10	Move backward one logical record
11	Move backward over segment mark
12	Rewind

DOCUMENT REVIEW SHEET

TITLE: GE-625/635 JOVIAL User's Information

CPB #: (none)

FROM:

Name: ______

Position: ______

Address: ______

______

Comments concerning this publication are solicited for use in improving future editions. Please provide any recommended additions, deletions, corrections, or other information you deem necessary for improving this manual. The following space is provided for your comments.

COMMENTS: ______

Please cut along this line

NO POSTAGE NECESSARY IF MAILED IN U.S.A.
Fold on two lines shown on reverse side, staple, and mail.

FOLD

FIRST CLASS
PERMIT, No. 4332
PHOENIX, ARIZONA

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY

GENERAL ELECTRIC COMPANY
COMPUTER EQUIPMENT DEPARTMENT
13430 NORTH BLACK CANYON HIGHWAY
PHOENIX, ARIZONA - 85029

ATTENTION: ENGINEERING PUBLICATIONS STANDARDS B-90

FOLD

GENERAL ELECTRIC
COMPUTERS

GE-625/635 JOVIAL COMPILER PROGRAMMING REFERENCE MANUAL

ADVANCE INFORMATION

GENERAL ELECTRIC

XCPB-1201

GE-625/635
JOVIAL COMPILER
PROGRAMMING REFERENCE MANUAL

October 1965

GENERAL ELECTRIC

COMPUTER DEPARTMENT

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

LIST OF EXHIBITS

I. INTRODUCTION

A. Background

The purpose of this text is to provide the trained JOVIAL programmer with a complete, rigorous, and concise manual to be used as a reference in answering specific questions about the construction and meaning of elements of JOVIAL programs.

This manual is organized topically, proceeding from the most primitive to the most complex elements of the language, with I/O and direct operations treated at the end. Each topic is divided into two parts--construction and meaning.

Construction is indicated via a concise format described in subsection I.C. Meaning is covered via brief prose descriptions in terms of basic computer characteristics, other elements, or combinations of simpler elements.

Each page is annotated in the lower outside corner as to the topic covered, so that the programmer may "flip through" easily to a given topic.

An index of constructions is provided, referencing the page on which each construction is defined.

B. Computer-Oriented Features

JOVIAL is not entirely a computer-independent programming language. Certain features of the language depend for their interpretation on features of the machine for which programs are to be written. These features are summarized below for the GE-635 object machine:

1. Internal storage: fixed-length word, binary
2. Word size: 36 bits
3. Storage capacity: up to 256K words, using an 18-bit address
4. Fixed-point arithmetic: two's complement; 36-bit signed full word; 72-bit signed double word is programmed
5. Floating-point arithmetic: two's complement; 8-bit signed exrad, 28-bit signed signicand
6. Character representation: 6-bit encoding (see Exhibit 1, p. 14)
7. External storage: magnetic tapes, discs, card reader and punch, and printer

C. Construction and Meaning

The remainder of this text is presented in a particular format with each element of the language being described first as to the form of its construction and second as to its interpretation.

The method chosen for displaying the construction of the elements can best be described by example:

1. The construction

   numeral: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

   means that the element we chose to call a numeral is constructed as either a 0 or a 1 or a 2 , etc. The | separates alternative constructions. Any of the alternatives yields a correct construction of the element numeral.

2. The construction

   sign: $\boxed{\text{letter}}$ | $\boxed{\text{numeral}}$ | $\boxed{\text{mark}}$

   means that the element which we chose to call a sign is constructed of either an element called a letter or an element called a numeral or an element called a mark. Thus a sign may be constructed as an A , a B , a 0 , a 1 , a + , an = , etc. The box around the name of the element differentiates it from actual characters forming part of a construction.

3. The construction

   decimal number: $\boxed{\text{number}}$. $\boxed{\text{number}}$ | $\boxed{\text{number}}$. | . $\boxed{\text{number}}$

   means that the element we chose to call a decimal number is constructed in any one of the following ways:

   a. of a number element followed by a . followed by a number element

   b. of a number element followed by a .

   c. of a . followed by a number element

   Any number of consecutively written elements and/or actual characters can define the construction of another element.

4. The construction

$$\text{number: } \boxed{\text{numeral}}\ \Big|\ \boxed{\text{numeral}}\ \Big|$$

means that the element we chose to call a number is constructed of a numeral followed by zero or more numerals. The brackets enclose the part of the construction which may either be absent entirely or may appear any number of times consecutively.

For example, the construction

$$\text{subscript list: } \boxed{\text{subscript}}\ \Big|\ ,\ \boxed{\text{subscript}}\ \Big|$$

means that the element we chose to call a subscript list is constructed of one or more subscripts separated by commas.

5. We also could have defined the constructions in example 4 as

$$\text{number: } \boxed{\text{numeral}}\ |\ \boxed{\text{number}}\ \boxed{\text{numeral}}$$
$$\text{subscript list: } \boxed{\text{subscript}}\ |\ \boxed{\text{subscript list}}\ ,\ \boxed{\text{subscript}}$$

using recursive constructions. This has generally been avoided in the text for reasons of clarity, except that occasionally an element being defined will appear in the definition of some one of its parts. This is unavoidable in, for example, a variable being part of a numeric formula and a numeric formula being part of a variable (i. e., as a subscript); this merely indicates that subscripts can include subscripted variables.

6. Using several of these concepts, we have the construction

$$\text{octal number: O (} \boxed{\text{octal numeral}}\ \Big|\ \boxed{\text{octal numeral}}\ \Big|\ )$$

which means that the element we chose to call an octal number is constructed of an O followed by a (followed by one or more octal numerals followed by a) .

When an element is used in a construction but has not been defined in the preceding text, a reference is given to the subsection in which it is defined.

The method chosen for describing the meanings of the constructed elements is straightforward prose text. However, certain words and phrases used should be defined more fully:

1.	Implicit	Means that the object being discussed does not appear in the written program but is understood to exist, either actually or for purposes of explanation.
2.	Explicit	Means that the object being discussed does or is to appear in the written program.
3.	Effectively (the effect of)	Means that the operations being discussed are understood to take place implicitly. This particularly refers to phrases like "replaced during execution," where such replacement does not actually take place but is understood to take place.
4.	Quantity	Means an entity containing a value.
5.	Entity	Means some actual part of the object program which has an identity of its own, such as an instruction, a set of bits allocated to an item, and so forth.
6.	Value	Means the value yielded by a quantity or by operations on other values.
	Actual value	Means the value without nonsignificant high-order zeros.
	Type of value	Means the category of value concerned, whether the value represents an integer value, a truth value, etc.
7.	Invoke	Means to call into effect a set of operations; usually this means to "call" a subroutine.
8.	Must	Means in most contexts that if the rule is not followed the results of the specified operations are not defined.
9.	Not defined	Means that the results of the operations involved should be considered spurious and are not guaranteed to be meaningful.

10. Precision	Means the location of the binary point relative to the right end of the computer representation of the value concerned: a positive precision indicates that the binary point is located to the left; a negative precision, to the right. Thus, the numbers:

77.77_8 with a precision of +3 means a value of 77.70_8 and a computer representation of 777

77.77_8 with a precision of 0 means a value of 77.00_8 and a computer representation of 77

77.77_8 with a precision of -3 means a value of 70.00_8 and a computer representation of 7

II. SIGNS, VALUES, AND SYMBOLS

A. Signs

1. Construction

sign: $\boxed{\text{letter}}$ | $\boxed{\text{numeral}}$ | $\boxed{\text{mark}}$

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

numeral: 0| 1| 2| 3| 4| 5| 6| 7| 8| 9

mark: +| -| *| /| =| .| ,| '| (|)| $| #

2. Meaning

The alphabet of which JOVIAL symbols are constructed consists of 48 signs. These signs do not have individual meanings, but are used to form symbols. Other signs not already in the alphabet may be included as additional marks. Such signs would be those permitted by the hardware of the computer. The # mark is the explicit denotation (in this document) of a single blank space.

B. Delimiters

1. Construction

delimiter: $\boxed{\text{operator}}$ | $\boxed{\text{modifier}}$ | $\boxed{\text{separator}}$ | $\boxed{\text{bracket}}$ |
$\boxed{\text{declarator}}$ | $\boxed{\text{descriptor}}$

operator: $\boxed{\text{arithmetic operator}}$ | $\boxed{\text{relational operator}}$ |
$\boxed{\text{logical operator}}$ | $\boxed{\text{sequential operator}}$ |
$\boxed{\text{peripheral operator}}$ | 'ASSIGN

arithmetic operator: + | - | * | / | **

relational operator: 'EQ | 'NQ | 'GR | 'LS | 'GQ | 'LQ

logical operator: 'AND | 'OR | 'NOT

sequential operator: 'IF | 'IFEITH | 'ORIF | 'FOR | 'TEST |
'GOTO | 'RETURN | 'STOP

peripheral operator: IN | OUT | IO $\boxed{\text{number}}$ | WAIT

modifier: 'BIT | 'BYTE | 'NENT | 'ENTRY | 'ENT | 'ABS |
'CHAR | 'MANT | 'NWDSEN | 'ODD | 'ALL

separator: . | , | = | == | $ | ... | #

bracket: (|) | ($ | $) | (/ | /) | (* | *) | " | 'BEGIN | 'END |
'DIRECT | 'JOVIAL | 'START | 'TERM

declarator: 'ITEM | 'MODE | 'ARRAY | 'TABLE | 'OVERLAY |
'SWITCH | 'CLOSE | 'PROC | 'FILE | 'DEFINE

descriptor: $\boxed{\text{type descriptor}}$ | $\boxed{\text{sign descriptor}}$ |
$\boxed{\text{rounding descriptor}}$ | $\boxed{\text{preset descriptor}}$ |
$\boxed{\text{length descriptor}}$ | $\boxed{\text{structure descriptor}}$ |
$\boxed{\text{packing descriptor}}$ | $\boxed{\text{pattern descriptor}}$

type descriptor: I | A | F | B | H | T | S sign descriptor: S | U

rounding descriptor: R

preset descriptor: P

length descriptor: V | R

structure descriptor: S | P

packing descriptor: N | M | D

pattern descriptor: L

2. Meaning

Delimiters have fixed meanings that are described later in the text. All delimiters shown above constructed with a leading prime, except for " , may also be constructed without the leading prime. These delimiters will be shown elsewhere in the text without the leading prime.

C. Values--Meaning

There are four primary types of values in JOVIAL: numeric, Boolean, literal, and status. Numeric values may further be categorized into integer, fixed, and floating values. Literal values may further be categorized into Hollerith and transmission values.

Integer values are integer numbers carried in fixed-point format in one or two words with zero precision. Their unsigned magnitude, m , must be

$$2^{71} - 1 \geq m \geq 0 \qquad (2^{71} > 10^{21})$$

Fixed values are decimal numbers carried in fixed-point format, in one or two words. A fixed value has a given precision, p , that determines its computer representation.

The unsigned magnitude, m , of a fixed value must be such that

$$2^{71} - 1 \geq \left[m \cdot 2^{p}\right] \geq 2^{-71} - 1 \quad \text{or} \quad \left[m \cdot 2^{p}\right] = 0$$

where the brackets indicate truncation of any fractional value.

Floating values are decimal numbers carried in floating point format in one word. Their unsigned magnitude, m , must be confined to the following range:

$$2^{-129} \leq m \leq (1 - 2^{-27})\, 2^{127}$$

$$m = 0 \qquad (2^{127} > 10^{38})$$

$$-(1 + 2^{-26})\, 2^{-129} \geq m \geq -2^{127}$$

The computer representation of floating values has a precision of 27 bits (i. e., about 8 significant digits).

Boolean values are the truth values, true and false, carried as a single bit.

Literal values are computer representations of strings of signs. One or more signs (i. e., bytes) may be included in the string. Hollerith values employ one type of six-bit encoding for the signs, and transmission values employ another. Literal values are carried in as many words as necessary to hold the packed strings. Exhibit 1 displays the six-bit encoding for both types of literal values.

Status values are members of ordered sets of unsigned integer values. Such a set may contain one or more members. The first member of a set is the value 0; the second, 1; the third, 2; and so forth. There may be up to 2^{71} members, and the values are carried as integer values although referenced symbolically.

Hollerith Encoding

Second Octal Digit

First Octal Digit	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	[	#	@	:	>	?
2	ƀ(1)	A	B	C	D	E	F	G
3	H	I	&	.	]	(	<	\
4	↑	J	K	L	M	N	O	P
5	Q	R	-	$	*	(	;	'
6	+	/	S	T	U	V	W	X
7	X	Z	←	,	%	=	"	!

Transmission Encoding

Second Octal Digit

First Octal Digit	0	1	2	3	4	5	6	7
0	%	↑	↓	[	]	ƀ(1)	A	B
1	C	D	E	F	G	H	I	J
2	K	L	M	N	O	P	Q	R
3	S	T	U	V	W	X	Y	Z
4	(	-	+	<	=	>	≤	$
5	*	(	≥	:	⌉	^	,	≠
6	0	1	2	3	4	5	6	7
7	8	9	∨	;	/	.	→	≡

Note: (1) ƀ indicates a blank space.

EXHIBIT 1 - ENCODING OF SIGNS

D. <u>Constants</u>

1. <u>Construction</u>

constant: $\boxed{\text{numeric constant}}$ | $\boxed{\text{Boolean constant}}$ |
$\boxed{\text{literal constant}}$ | $\boxed{\text{status constant}}$

numeric constant: $\boxed{\text{integer constant}}$ | $\boxed{\text{fixed constant}}$ |
$\boxed{\text{floating constant}}$

integer constant: $\boxed{\text{number}}$ E $\boxed{\text{number}}$ | $\boxed{\text{number}}$ |
$\boxed{\text{octal number}}$

fixed constant: $\boxed{\text{decimal number}}$ E $\boxed{\text{scale}}$ A $\boxed{\text{precision}}$ |
$\boxed{\text{decimal number}}$ A $\boxed{\text{precision}}$

floating constant: $\boxed{\text{decimal number}}$ E $\boxed{\text{scale}}$ |
$\boxed{\text{decimal number}}$

decimal number: $\boxed{\text{number}}$. $\boxed{\text{number}}$ | $\boxed{\text{number}}$. |
. $\boxed{\text{number}}$

scale: + $\boxed{\text{number}}$ | - $\boxed{\text{number}}$ | $\boxed{\text{number}}$

precision: + $\boxed{\text{number}}$ | - $\boxed{\text{number}}$ | $\boxed{\text{number}}$

number: $\boxed{\text{numeral}}$ | $\boxed{\text{numeral}}$ |

octal number: O ($\boxed{\text{octal numeral}}$ | $\boxed{\text{octal numeral}}$ |)

octal numeral: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Boolean constant: 0 | 1 | FALSE | TRUE

literal constant: $\boxed{\text{Hollerith constant}}$ | $\boxed{\text{transmission constant}}$

Hollerith constant: $\boxed{\text{number}}$ H ($\boxed{\text{sign}}$ | $\boxed{\text{sign}}$ |) |
$\boxed{\text{octal number}}$

transmission constant: $\boxed{\text{number}}$ T ($\boxed{\text{sign}}$ | $\boxed{\text{sign}}$ |) |
$\boxed{\text{octal number}}$

status constant: V ($\boxed{\text{identifier}}$) | V ($\boxed{\text{letter}}$)

identifier: $\boxed{\text{letter}}$ $\boxed{\text{part of identifier}}$ $\Big|$ $\boxed{\text{part of identifier}}$ $\Big|$

part of identifier: $\boxed{\text{letter}}$ | $\boxed{\text{numeral}}$ | ' $\boxed{\text{letter}}$ |

' $\boxed{\text{numeral}}$

2. Meaning

Integer constants are explicit representations of unsigned integer values. The elements E $\boxed{\text{number}}$ are interpreted as meaning

$$mEn = m \cdot 10^{n}$$

where m is the first number and n is the number following the E .

Fixed constants are explicit representations of unsigned, fixed values. The elements E $\boxed{\text{scale}}$ are interpreted as meaning

$$mEs = m \cdot 10^{s}$$

where m is the decimal number and s is the scale. The elements A $\boxed{\text{precision}}$ establish the precision of the constant.

Floating constants are explicit representations of unsigned floating values. The elements E $\boxed{\text{scale}}$ are interpreted as in fixed constants.

Boolean constants are explicit representations of Boolean values; 0 is taken to mean false and 1 to mean true.

Hollerith constants are explicit representations of Hollerith values. The signs enclosed in parentheses are the signs represented. The number preceding the H must be an exact count of these signs. An octal number used as a Hollerith constant must specify 24 or less octal numerals which compose proper Hollerith signs.

Transmission constants are explicit representations of transmission values. The signs enclosed in parentheses are the signs represented. The number preceding the T must be an exact count of the signs. An octal number used as a transmission constant must specify 24 or less octal numerals which compose proper transmission signs.

Status constants are explicit, though symbolic, representations of status values. Each value in a set of status values (see subsection VI.A, Item Descriptions) is denoted by a unique identifier or letter, whose correspondence to the unsigned integer value is obtained by declaration and context.

Each of the above type of constant must be written to represent a value within the limitations of its type as specified in the last subsection.

E. Names and Indexes

1. Construction

name: $\boxed{\text{set name}}$ | $\boxed{\text{goto name}}$

set name: $\boxed{\text{data name}}$ | $\boxed{\text{procedure name}}$ | $\boxed{\text{file name}}$

data name: $\boxed{\text{simple item name}}$ | $\boxed{\text{array name}}$ | $\boxed{\text{table name}}$ | $\boxed{\text{table item name}}$

goto name: $\boxed{\text{statement name}}$ | $\boxed{\text{switch name}}$ | $\boxed{\text{close name}}$

switch name: $\boxed{\text{index switch name}}$ | $\boxed{\text{item switch name}}$ | $\boxed{\text{file switch name}}$

simple item name: $\boxed{\text{identifier}}$

array name: $\boxed{\text{identifier}}$

table name: $\boxed{\text{identifier}}$

table item name: $\boxed{\text{identifier}}$

procedure name: $\boxed{\text{identifier}}$

file name: $\boxed{\text{identifier}}$

statement name: $\boxed{\text{identifier}}$

index switch name: $\boxed{\text{identifier}}$

item switch name: $\boxed{\text{identifier}}$

file switch name: $\boxed{\text{identifier}}$

close name: $\boxed{\text{identifier}}$

index: $\boxed{\text{letter}}$

2. Meaning

Names serve to identify the entities of a program environment: simple items, arrays, tables, table items, procedures, files, statements, switches, and close routines.

No identifier used for a name may be constructed identically to any delimiter (GOTO, IN, LQ) or constant (TRUE, FALSE). Procedure names must not contain more than six characters.

For purposes of reference by machine codes, names are associated with entities of the object program as noted below:

a. Simple item name--first word of simple item
b. Array name--first word in array
c. Table name--first word in table, excluding "nent word"
d. Table item name--first word in which first instance of table item occurs
e. Procedure name--first executable instruction in procedure
f. File name--first word of corresponding status value
g. Statement name--first executable instruction of statement
h. Switch name--first executable instruction of switch routine
i. Close name--first executable instruction of close routine

Indexes serve to identify members of an implicit set of integer-valued quantities, which are primarily applied in controlling iteration of JOVIAL operations.

The scope of the definition of names and indexes is described in subsection VIII.B, Scope of Definitions.

F. <u>Symbols and Spacing</u>

1. <u>Construction</u>

symbol: $\boxed{\text{delimiter}}$ | $\boxed{\text{constant}}$ | $\boxed{\text{name}}$ | $\boxed{\text{index}}$ |
$\boxed{\text{comment}}$ | $\boxed{\text{device code}}$ | $\boxed{\text{machine code}}$ |
$\boxed{\text{defined identifier}}$ | $\boxed{\text{accumulator designation}}$

comment: " $\left|\boxed{\text{part of comment}}\right|$ "

part of comment: $\boxed{\text{comment sign}}$ | ' $\boxed{\text{comment sign}}$

comment sign: $\boxed{\text{letter}}$ | $\boxed{\text{numeral}}$ | + | - | * | / | = | . | , | (|
) | ($ | $) | #

logical file code: see subsection IX.A, File Declarations and Status

machine code: see subsection X.A, Direct Statements

defined identifier: see subsection VIII.C, Define Declarations

accumulator designation: see subsection IX.B, Assign Statements

2. <u>Meaning</u>

Symbols are the words and punctuation from which JOVIAL programs are constructed. A symbol cannot contain any embedded spaces except where explicitly permitted; spaces are only explicitly permitted as a separator and within literal constants, comments, and machine codes.

Where a space may be required between symbols, it will be shown explicitly in the remainder of the text. Where it is shown, it is required only if the following sign is a letter, numeral, prime, or decimal point. Otherwise, any number of spaces may or may not appear arbitrarily between symbols.

A comment may appear in place of any such arbitrary space within a program, except between the identifier and the following " in a define declaration. The effect of a comment is exactly that of an arbitrary space.

III. VARIABLES, FUNCTIONS, AND FORMULAS

A. Variables

1. Construction

variable: $\boxed{\text{numeric variable}}$ | $\boxed{\text{Boolean variable}}$ |

$\boxed{\text{literal variable}}$ | $\boxed{\text{status variable}}$

numeric variable: $\boxed{\text{integer variable}}$ | $\boxed{\text{floating variable}}$ |

$\boxed{\text{fixed variable}}$

integer variable: $\boxed{\text{basic variable}}$ | $\boxed{\text{index}}$ | $\boxed{\text{bit variable}}$ |

$\boxed{\text{nent variable}}$ | $\boxed{\text{char variable}}$

floating variable: $\boxed{\text{basic variable}}$

fixed variable: $\boxed{\text{basic variable}}$ | $\boxed{\text{mant variable}}$

Boolean variable: $\boxed{\text{basic variable}}$ | $\boxed{\text{odd variable}}$

literal variable: $\boxed{\text{Hollerith variable}}$ | $\boxed{\text{transmission variable}}$

Hollerith variable: $\boxed{\text{basic variable}}$ | $\boxed{\text{byte variable}}$

transmission variable: $\boxed{\text{basic variable}}$ | $\boxed{\text{byte variable}}$

status variable: $\boxed{\text{basic variable}}$

basic variable: $\boxed{\text{simple variable}}$ | $\boxed{\text{subscripted variable}}$

simple variable: $\boxed{\text{simple item name}}$

subscripted variable: $\boxed{\text{array name}}$ (\$ $\boxed{\text{subscript list}}$ \$) |

$\boxed{\text{table item name}}$ (\$ $\boxed{\text{subscript}}$ \$)

subscript list: $\boxed{\text{subscript}} \left| \, , \boxed{\text{subscript}} \right|$

subscript: $\boxed{\text{numeric formula}}$

bit, byte, nent, char, mant, and odd variables: see next subsection, Special Variables

numeric formula: see subsection III.E, Numeric Formulas

2. Meaning

A variable serves to identify a quantity which may be used in formulas and elsewhere to yield a value, and whose value may be changed by assignment and other statements.

The type of value assignable to a particular basic variable is defined in the declaration for the simple item name, array name, or table item name; the type of value assignable to a particular nonbasic (i. e., special) variable is indicated by its construction (see next subsection). Integer variables yield and are assigned only integer values; fixed variables, fixed values; floating variables, floating values; Boolean variables, Boolean values; Hollerith variables, Hollerith values; transmission variables, transmission values; and status variables, status values. A subscripted variable identifies a quantity which is either an instance of a table item or an element of an array. The value of a subscript to a table item name designates the specific instance of the item desired, where 0 designates the first instance; 1, the second; 2, the third; and so forth.

The values of the subscripts to an array name designate the specific element of the array desired, where the first subscript from the left designates the specific row in which the element appears; the second, the specific column; the third, the specific matrix; and so forth. A 0 value yielded by a subscript designates the first set of elements in that dimension; a 1 value, the second set; and so forth.

If the value yielded by a subscript is not an integer value, it is truncated to an integer value. That value must then be no less than zero nor greater than the declared length of the table or the declared size of the corresponding array dimension.

The value of any variable is undefined until a value is explicitly assigned (e. g., by "preset," by assignment statement, by procedure, etc.).

An index is implicitly declared as I 18 S.

B. <u>Special Variables</u>

1. <u>Construction</u>

bit variable: BIT ($ $\boxed{\text{bit designation}}$ $) ($\boxed{\text{basic variable}}$)

bit designation: $\boxed{\text{first bit}}$, $\boxed{\text{number of bits}}$ | $\boxed{\text{first bit}}$

first bit: $\boxed{\text{numeric formula}}$

number of bits: $\boxed{\text{numeric formula}}$

byte variable: BYTE ($ $\boxed{\text{byte designation}}$ $)

($\boxed{\text{basic variable}}$)

byte designation: $\boxed{\text{first byte}}$, $\boxed{\text{number of bytes}}$ | $\boxed{\text{first byte}}$

first byte: $\boxed{\text{numeric formula}}$

number of bytes: $\boxed{\text{numeric formula}}$

nent variable: NENT ($\boxed{\text{table designation}}$)

table designation: $\boxed{\text{table name}}$ | $\boxed{\text{table item name}}$

char variable: CHAR ($\boxed{\text{basic variable}}$)

mant variable: MANT ($\boxed{\text{basic variable}}$)

odd variable: ODD ($\boxed{\text{basic variable}}$) | ODD ($\boxed{\text{index}}$)

2. <u>Meaning</u>

The bit variable serves to identify a segment of the bit string comprising the basic variable as the bit variable. The basic variable may be of any type. The bit designation specifies the first bit in the segment, and if more than one contiguous bit is in the segment, the number of the bits.

The bits in the basic variable are numbered consecutively from left to right beginning with zero. The first bit is designated using this number. The designated bits must form a subset of the bit string of the basic variable. The values of the numeric formulas, if not integer values, are truncated to integer values and must be no less than zero. The value of the bit variable is considered to be an unsigned integer value.

The byte variable serves to identify a segment of the byte string making up the basic variable as the byte variable. The basic variable must be of a literal type. The byte designation specifies the included bytes in a manner completely analogous to bit designation for bit variables. The value of the byte variable is considered to be the same type as the basic variable.

The nent variable serves to identify the nent word of a variable-length table as the nent variable. The table designated is the one named or the one containing the named table item. The value of the nent variable is considered to be an unsigned integer value, whose magnitude must not exceed that of the declared maximum number of entries for the designated table.

The char variable serves to identify the exrad of the basic variable. Thus, the basic variable must be a floating variable. The value of the char variable is considered to be an integer value between -127 and +127, inclusive; i. e., implicitly declared as I 8 S.

The mant variable serves to identify the signicand of the basic variable. Thus, the basic variable must be a floating variable. The value of the mant variable is considered to be a fixed value less than or equal to $1 - 2^{27}$ and greater than or equal to -2^{-2}, and has a precision of i. e., implicitly declared as A 28 S 27.

The odd variable serves to identify the least significant bit of the basic variable. The basic variable must be a numeric variable. Fixed and floating variables are treated as though their bit pattern constituted an integer variable. The value of the odd variable is false if the bit is a 0, and the value is true if the bit is a 1.

C. Functions

1. Construction

function: $\boxed{\text{numeric function}}$ | $\boxed{\text{Boolean function}}$ |

$\boxed{\text{literal function}}$ | $\boxed{\text{status function}}$

numeric function: $\boxed{\text{integer function}}$ | $\boxed{\text{fixed function}}$ |

$\boxed{\text{floating function}}$ | $\boxed{\text{abs function}}$

integer function: $\boxed{\text{basic function}}$ | $\boxed{\text{nent function}}$ |

$\boxed{\text{nwdsen function}}$

fixed function: $\boxed{\text{basic function}}$

floating function: $\boxed{\text{basic function}}$

Boolean function: $\boxed{\text{basic function}}$

literal function: $\boxed{\text{Hollerith function}}$ | $\boxed{\text{transmission function}}$

Hollerith function: $\boxed{\text{basic function}}$

transmission function: $\boxed{\text{basic function}}$

status function: $\boxed{\text{basic function}}$

basic function: $\boxed{\text{procedure name}}$ ($\boxed{\text{actual input parameter list}}$)

abs, nent, and nwdsen functions: see next subsection,

Special Functions

actual input parameter list: see subsection IV.E, Procedure

Statements

2. Meaning

A function serves to invoke the execution of a procedure which yields a value that may be used in formulas. The value of a basic function results from the execution of the set of operations defined by the procedure declaration with the same procedure name. The value of a nonbasic (special) function results from the execution of certain implicit sets of operations (see next subsection).

The type of value yielded by a basic function is defined in the declaration for the simple item within the procedure which has the same name as the procedure; the type of value yielded by a special function is indicated by its construction.

The construction and meaning of actual input parameter lists are given in subsection IV.E, Procedure Statements.

The value of a literal function must not exceed 12 bytes.

D. Special Functions

1. Construction

abs function: ABS ($\boxed{\text{numeric formula}}$)

nent function: NENT ($\boxed{\text{table designation}}$)

nwdsen function: NWDSEN ($\boxed{\text{table designation}}$)

2. Meaning

The abs function serves to identify the value of the unsigned magnitude of the numeric formula. The value of the abs function is considered to be of the same type as the value yielded by the numeric formula.

The nent function serves to identify the value of the nent word of the designated rigid-length table. The value of the nent function is considered to be an unsigned integer value.

The nwdsen function serves to identify a value that is equal to the number of computer words occupied by an entry of the designated table. The value of the nwdsen function is considered to be an unsigned integer value.

E. Numeric Formulas

1. Construction

numeric formula: $\boxed{\text{term}} \left| \boxed{\text{add operator}} \; \boxed{\text{term}} \right|$

add operator: + | -

term: $\boxed{\text{factor}} \left| \boxed{\text{multiply operator}} \; \boxed{\text{factor}} \right|$

multiply operator: * | /

factor: $\boxed{\text{secondary}} \left| \boxed{\text{exponent}} \right|$

exponent: ** $\boxed{\text{secondary}}$ | (* $\boxed{\text{numeric formula}}$ *)

secondary: $\left| \boxed{\text{add operator}} \right| \boxed{\text{primary}}$

primary: $\boxed{\text{numeric constant}}$ | $\boxed{\text{numeric variable}}$ |
$\boxed{\text{numeric function}}$ | ($\boxed{\text{numeric formula}}$) |
(/ $\boxed{\text{numeric formula}}$ /)

2. Meaning

A numeric formula serves to identify a set of operations for computing a numeric value. The value is obtained by executing the indicated operations on the values specified within the numeric formula.

The value of numeric constants, variables, and functions has been described in other subsections. The value of a numeric formula enclosed in parentheses is simply the value of the enclosed numeric formula.

The value of a numeric formula enclosed in the brackets (/ /) can be interpreted as:

(/n/) = ABS(n)

where n is a numeric formula.

The value of a complex secondary is interpreted as:

. . . oop. . . = . . . 0o(0op). . .

where o is an add operator and p is a primary.

The value of a complex factor is interpreted as:

$$\ldots s(*n*)(*n*)(*n*)\ldots = \ldots s**(n)**(n)**(n)\ldots$$

$$\ldots s**s**s\ldots = \ldots(s**s)**s\ldots$$

where s is a secondary and n is a numeric formula.

The value of a complex term is interpreted as:

$$\ldots fofof\ldots = \ldots(fof)of\ldots$$

where f is a factor and o is a multiply operator.

The value of a complex numeric formula is interpreted as:

$$\ldots totot\ldots = \ldots(tot)ot\ldots$$

where t is a term and o is an add operator.

Unary or binary addition and subtraction are indicated by + and - , multiplication and division by * and / , and exponentiation by ** .

The constructions given in this subsection also indicate the order in which the operations are carried out. Thus,

a. A primary is evaluated before any secondary of which it is a part is evaluated.

b. A secondary is evaluated before any factor of which it is a part is evaluated.

c. A factor is evaluated before any term of which it is a part is evaluated.

d. A term is evaluated before any numeric formula of which it is a part is evaluated.

No other ordering of operations is implied by the foregoing constructions aside from that given in the interpretations above.

Thus, if a function occurring in a numeric formula has side effects involving other quantities in the formula (such as changing the value of some variable), the value of the formula is not defined.

The type and precision of the value yielded by a numeric formula is determined by the type and precision of the values of the constituent constants, variables, and functions. This determination is made according to the following rules:

a. If the value of any constituent is a floating value, the result will be a floating value.

b. If not item a, but any constituent is a double-word fixed value, the result will be a double-word fixed value; its precision will be that of the least precise fixed constituent after increasing the precision of any single-word fixed constituent by 35 places.

c. If not a or b, but any constituent is a double-word integer value and any other constituent is a single-word fixed value, the result will be a double-word fixed value; its precision will be that of the least precise fixed constituent.

d. If not a, b, or c, but any constituent is a single-word fixed value, the result will be a single-word fixed value; its precision will be that of the least precise fixed constituent.

e. If not a, b, c, or d, all constituents are integer values, and the result is an integer value--double-word if any constituent is double-word and single-word otherwise.

Note that a 36-bit unsigned value is considered a double-word value in numeric formulas.

All constituents are converted to the type and precision determined for the result, and all operations are performed in that type and precision. Least significant bits lost due to conversion are truncated; any added are zero valued.

If the actual magnitude of the result exceeds 35 bits (for single-word fixed or integer results) or 71 bits (for double-word fixed or integer results), the result is not defined.

Certain operations on particular values are not defined. These are as follows:

a/b	where b = 0
a**b	where a = 0
a**b	where a < 0 and b is not an integer value

F. Boolean Formulas

1. Construction

Boolean formula: $\boxed{\text{Boolean term}}$ | #OR# $\boxed{\text{Boolean term}}$ |

Boolean term: $\boxed{\text{Boolean secondary}}$ | #AND#
$\boxed{\text{Boolean secondary}}$ |

Boolean secondary: | #NOT# | $\boxed{\text{Boolean primary}}$

Boolean primary: $\boxed{\text{Boolean constant}}$ | $\boxed{\text{Boolean variable}}$ |
$\boxed{\text{Boolean function}}$ | ($\boxed{\text{Boolean formula}}$) | $\boxed{\text{relation}}$

relation: $\boxed{\text{numeric relation}}$ | $\boxed{\text{literal relation}}$ |
$\boxed{\text{status relation}}$ | $\boxed{\text{entry relation}}$ | $\boxed{\text{file status relation}}$

numeric relation: $\boxed{\text{numeric formula}}$ # $\boxed{\text{relational operator}}$ #
$\boxed{\text{numeric formula}}$ | # $\boxed{\text{relational operator}}$ #
$\boxed{\text{numeric formula}}$ |

literal relation: $\boxed{\text{literal formula}}$ # $\boxed{\text{relational operator}}$ #
$\boxed{\text{literal formula}}$ | # $\boxed{\text{relational operator}}$ #
$\boxed{\text{literal formula}}$ |

status relation: $\boxed{\text{status variable}}$ # $\boxed{\text{relational operator}}$ #
$\boxed{\text{status formula}}$

file status relation: see subsection IX.A, File Declarations and Status

entry relation: see subsection IV.C, Entry Operations

literal and status formula: see next subsection, Literal and Status Formulas

2. Meaning

A Boolean formula serves to identify a set of operations for computing a truth value. The value is obtained by executing the indicated operations on the values specified within the Boolean formula.

The value of Boolean constants, variables, and functions has been described in other subsections. The value of a Boolean formula in parentheses is simply the value of the enclosed Boolean formula.

The value of a complex Boolean primary is interpreted as:

$$\ldots n_1 o n_2 o n_3 \ldots = \ldots (n_1 o n_2 \text{ AND } n_2 o n_3) \ldots$$

where n is a numeric (or literal) formula and o is a relational operator.

The value of a complex Boolean secondary is interpreted as:

$$\ldots \text{NOT NOT } p \ldots = \ldots \text{NOT (NOT } p) \ldots$$

where p is a Boolean primary.

The value of a complex Boolean term is interpreted as:

$$\ldots s \text{ AND } s \text{ AND } s \ldots = \ldots (s \text{ AND } s) \text{ AND } s \ldots$$

where s is a Boolean secondary.

The value of a complex Boolean formula is interpreted as:

$$\ldots t \text{ OR } t \text{ OR } t \ldots = \ldots (t \text{ OR } t) \text{ OR } t \ldots$$

where t is a Boolean term.

The Boolean formula t OR t , where t is a Boolean term, yields the value false if both t's are false, and otherwise yields the value true. The Boolean term s AND s , where s is a Boolean secondary, yields the value true if both s's are true, and otherwise yields the value false. The Boolean secondary NOT p , where p is a Boolean primary, yields the value true if p is false, and the value false if p is true.

The meaning of the relational operators is as follows:

EQ	Equal to
NQ	Not equal to
GR	Greater than
LS	Less than
GQ	Greater than or equal to
LQ	Less than or equal to

The constructions given in this subsection also indicate the order in which the operations are carried out. Thus:

a. A numeric formula is evaluated before any Boolean primary of which it is a part is evaluated.

b. A Boolean primary is evaluated before any Boolean secondary of which it is a part is evaluated.

c. A Boolean secondary is evaluated before any Boolean term of which it is a part is evaluated.

d. A Boolean term is evaluated before any Boolean formula of which it is a part is evaluated.

No other ordering of operations is implied by the foregoing constructions, aside from that given in the interpretations above.

Thus, if a function occurring in a Boolean formula has side effects involving other quantities in the formula, the value of the Boolean formula is not defined. Furthermore, only enough operations are evaluated to establish the truth value of the formula.

The value resulting from the evaluation of a relation is determined by comparing the two values involved. These comparisons are made in the following manner:

a. For numeric formulas: algebraically.

b. For literal formulas: by considering both values as unsigned integer values of unlimited magnitude. If the values are of unequal size, the shorter is considered to be prefixed with enough of the following signs to match the size of the longer value:

   (1) Hollerith formula: prefix Hollerith blanks.

   (2) Transmission formula: prefix transmission blanks.

The result of a comparison between literal values of different types is not defined. Comparison of two octal numbers is made algebraically.

c. For status variables and formulas: by numerically comparing the unsigned integer values which each represents. The status formula must yield a value representing an unsigned integer value which is a value also representable by the status variable.

G. Literal and Status Formulas

1. Construction

literal formula: $\boxed{\text{Hollerith formula}}$ | $\boxed{\text{transmission formula}}$

Hollerith formula: $\boxed{\text{Hollerith constant}}$ | $\boxed{\text{Hollerith variable}}$ |

$\boxed{\text{Hollerith function}}$

transmission formula: $\boxed{\text{transmission constant}}$ |

$\boxed{\text{transmission variable}}$ | $\boxed{\text{transmission function}}$

status formula: $\boxed{\text{status constant}}$ | $\boxed{\text{status variable}}$ |

$\boxed{\text{status function}}$

2. Meaning

A literal formula serves to identify a literal value. A status formula serves to identify a status value. A status constant has meaning only if it appears in context with a variable for which it is a declared value. Such context may be provided in a status relation, status assignment statement, or item switch declaration.

IV. SIMPLE STATEMENTS

A. Assignment Statements

1. Construction

assignment statement: $\boxed{\text{numeric assignment statement}}$ |

$\boxed{\text{Boolean assignment statement}}$ |

$\boxed{\text{literal assignment statement}}$ |

$\boxed{\text{status assignment statement}}$ |

$\boxed{\text{entry assignment statement}}$

numeric assignment statement: $\boxed{\text{numeric variable}}$ =

$\boxed{\text{numeric formula}}$ $

Boolean assignment statement: $\boxed{\text{Boolean variable}}$ =

$\boxed{\text{Boolean formula}}$ $

literal assignment statement: $\boxed{\text{literal variable}}$ =

$\boxed{\text{literal formula}}$ $

status assignment statement: $\boxed{\text{status variable}}$ =

$\boxed{\text{status formula}}$ $

entry assignment statement: see subsection IV.C, Entry Operations

2. Meaning

An assignment statement serves to specify the assignment of a value to a variable.

The type, precision, and/or structure of the value may be altered upon being assigned to the variable according to the following rules:

a. For numeric assignment statements:

(1) A floating value assigned to a floating variable remains unaltered.

(2) A fixed or integer value assigned to a floating variable will be converted to a floating value with the precision of a floating value.

(3) A floating value assigned to a fixed or integer variable will be converted to a fixed or integer value, respectively, with the precision of the variable.

(4) A fixed or integer value assigned to a fixed or integer variable will be converted to the precision of the variable.

(5) Least significant bits lost due to a change in the precision of a value are truncated or rounded as indicated by the declaration pertaining to the variable.

(6) Least significant bits added due to such a change are zero valued.

(7) The magnitude of the actual value must not exceed the maximum permitted by the declaration pertaining to the variable.

(8) A negative value must not be assigned to a variable declared to be unsigned.

b. For literal values:

(1) A literal value assigned to a literal variable which is shorter than the size of the value will cause the excess bytes to be truncated from the left end of the value.

(2) A literal value assigned to a literal variable which is longer than the size of the value will cause sufficient bytes to be prefixed to the value to match the size of the variable. If the variable is Hollerith, Hollerith blanks will be prefixed; if the variable is transmission, transmission blanks will be prefixed.

(3) A literal value must not be assigned to a literal variable of a different type.

c. For status values: The value assigned must represent an unsigned integer value which is also a value representable by the variable.

B. Exchange Statements

1. Construction

exchange statement: $\boxed{\text{numeric exchange statement}}$ |

$\boxed{\text{Boolean exchange statement}}$ |

$\boxed{\text{literal exchange statement}}$ |

$\boxed{\text{status exchange statement}}$ | $\boxed{\text{entry exchange statement}}$

numeric exchange statement: $\boxed{\text{numeric variable}}$ ==

$\boxed{\text{numeric variable}}$ $

Boolean exchange statement: $\boxed{\text{Boolean variable}}$ ==

$\boxed{\text{Boolean variable}}$ $

literal exchange statement: $\boxed{\text{literal variable}}$ ==

$\boxed{\text{literal variable}}$ $

status exchange statement: $\boxed{\text{status variable}}$ ==

$\boxed{\text{status variable}}$ $

entry exchange statement: see next subsection, Entry Operations

2. Meaning

An exchange statement serves to specify the assignment of the value of two variables to each other.

The effect of an exchange statement may be expressed in terms of simpler statements. The exchange statement a == b$ has the effect of being replaced during execution by

```
t = a$
a = b$
b = t$
```

where a and b are variables and t is an implicit variable of the same type and structure as a .

C. Entry Operations

1. Construction

entry relation: $\boxed{\text{entry variable}}$ #EQ# $\boxed{\text{entry formula}}$ |

$\boxed{\text{entry variable}}$ #NQ# $\boxed{\text{entry formula}}$

entry assignment statement: $\boxed{\text{entry variable}}$ =

$\boxed{\text{entry formula}}$ \$

entry exchange statement: $\boxed{\text{entry variable}}$ ==

$\boxed{\text{entry variable}}$ \$

entry formula: $\boxed{\text{entry variable}}$ | 0

entry variable: ENTRY ($\boxed{\text{table designation}}$ (\$ $\boxed{\text{subscript}}$ \$)) |

ENT ($\boxed{\text{table designation}}$ (\$ $\boxed{\text{subscript}}$ \$))

2. Meaning

An entry variable (which is not otherwise considered a variable) serves to identify the agglomerated bit string derived by considering an entire entry of the designated table as a single quantity. The structure of the quantity is obviously dependent on the structure and relationships of the individual table items comprising an entry. The specific entry designated in the table is the one whose items are associated with the value of the subscript.

An entry formula (which is not otherwise considered a formula) serves to identify the agglomerated value specified by an entry variable, or else a single zero bit.

The value resulting from the evaluation of an entry relation (true or false) is determined by comparing the value of the entry variable and the value of the entry formula, considering each value to be an unsigned integer value of unlimited magnitude.

The assignment of the value of an entry formula to an entry variable is made by considering the value to be an unsigned integer value and the variable to be an unsigned integer variable, except that if the magnitude of the actual value exceeds the maximum indicated by the entry variable, excess bits are truncated from the left of the value.

The effect of an entry exchange statement may be expressed in terms of entry assignment statements in a manner analogous to the expression of other exchange statements in terms of other assignment statements.

D. Goto Statements

1. Construction

goto statement: GOTO# $\boxed{\text{sequence designation}}$

sequence designation: $\boxed{\text{statement name}}$ | $\boxed{\text{close name}}$ |

$\boxed{\text{index switch name}}$ (\$ $\boxed{\text{numeric formula}}$ \$) |

$\boxed{\text{item switch name}}$ | $\boxed{\text{item switch name}}$

(\$ $\boxed{\text{subscript list}}$ \$) | $\boxed{\text{file switch name}}$

2. Meaning

A goto statement serves to specify an interruption in the normal sequence in which operations are executed (i. e., the sequence in which they are written).

When a statement name follows the GOTO, the statement with that name prefixed to it is the next statement to be executed. When a close name follows the GOTO, the goto statement invokes the execution of the close declaration which defines that name (see subsection VII.B, Close Declarations). When a switch name follows the GOTO, the goto statement invokes the execution of the switch declaration which defines that name (see subsection VII.A, Switch Declarations).

E. Procedure Statements

1. Construction

procedure statement: $\boxed{\text{procedure name}}$

$($ $\boxed{\text{actual parameter list}}$ $)$ \$ | $\boxed{\text{procedure name}}$ \$

actual parameter list: $\boxed{\text{actual input parameter list}}$

$\boxed{\text{actual output parameter list}}$

actual input parameter list: $\boxed{\text{actual input parameter}}$

$\left| \, , \boxed{\text{actual input parameter}} \right|$ | #

actual output parameter list: = $\boxed{\text{actual output parameter}}$

$\left| \, , \boxed{\text{actual output parameter}} \right|$ | #

actual input parameter: $\boxed{\text{formula}}$ | $\boxed{\text{array name}}$ |

$\boxed{\text{table name}}$ | $\boxed{\text{close name}}$.

actual output parameter: $\boxed{\text{variable}}$ | $\boxed{\text{array name}}$ |

$\boxed{\text{table name}}$ | $\boxed{\text{statement name}}$.

formula: $\boxed{\text{numeric formula}}$ | $\boxed{\text{Boolean formula}}$ |

$\boxed{\text{literal formula}}$ | $\boxed{\text{status formula}}$

2. Meaning

A procedure statement serves to invoke the execution of the procedure declaration which defines that name (see subsection VII.D, Procedure Declarations).

An actual input parameter must not be a status constant.

F. Stop Statements

1. Construction

stop statement: STOP $

2. Meaning

A stop statement serves to specify the operational end of a program.

G. Compound Statements

1. Construction

compound statement: BEGIN# $\boxed{\text{sentence list}}$ END# |

$\boxed{\text{direct statement}}$

sentence list: | $\boxed{\text{sentence}}$ | $\boxed{\text{statement}}$ | $\boxed{\text{sentence}}$ |

sentence: $\boxed{\text{statement}}$ | $\boxed{\text{declaration}}$

statement: $\boxed{\text{simple statement}}$ | $\boxed{\text{complex statement}}$

simple statement: $\boxed{\text{assignment statement}}$ |

$\boxed{\text{exchange statement}}$ | $\boxed{\text{goto statement}}$ |

$\boxed{\text{procedure statement}}$ | $\boxed{\text{stop statement}}$ |

$\boxed{\text{compound statement}}$ | $\boxed{\text{name}}$. $\boxed{\text{simple statement}}$ |

$\boxed{\text{test statement}}$ | $\boxed{\text{return statement}}$ | $\boxed{\text{in statement}}$ |

$\boxed{\text{out statement}}$ | $\boxed{\text{ion statement}}$ | $\boxed{\text{wait statement}}$

complex statement: $\boxed{\text{if statement}}$ | $\boxed{\text{ifeither statement}}$ |

$\boxed{\text{for statement}}$ | $\boxed{\text{name}}$. $\boxed{\text{complex statement}}$

declaration: $\boxed{\text{data declaration}}$ | $\boxed{\text{process declaration}}$ |

$\boxed{\text{file declaration}}$ | $\boxed{\text{define declaration}}$

data declaration: $\boxed{\text{simple item declaration}}$ |

$\boxed{\text{mode declaration}}$ | $\boxed{\text{array declaration}}$ |

$\boxed{\text{table declaration}}$ | $\boxed{\text{independent overlay declaration}}$

table declaration: $\boxed{\text{ordinary table declaration}}$ |

$\boxed{\text{specified table declaration}}$ | $\boxed{\text{like table declaration}}$

process declaration: $\boxed{\text{switch declaration}}$ | $\boxed{\text{close declaration}}$ |

$\boxed{\text{procedure declaration}}$ | $\boxed{\text{program declaration}}$

switch declaration: $\boxed{\text{index switch declaration}}$ |

$\boxed{\text{item switch declaration}}$ | $\boxed{\text{file switch declaration}}$

2. Meaning

A compound statement serves to specify a set of statements to be employed as a simple statement. At least one statement must be included within a compound statement; any number of declarations also may be included, but no special significance is attached to the fact that they are included. A direct statement is considered a compound statement.

The appearance of a name prefixed to a simple or complex statement serves to define the name as a statement name, and the following statement as the entity to which the name refers.

A statement name defined within a for statement must not be referenced from outside that for statement.

V. COMPLEX STATEMENTS

A. If Statements

1. Construction

if statement: $\boxed{\text{if clause}}$ $\boxed{\text{simple statement}}$

if clause: IF# $\boxed{\text{Boolean formula}}$ \$

2. Meaning

An if statement serves to specify the conditional execution of a simple statement.

The effect of an if statement is as follows: if the value of the Boolean formula is true, the simple statement is executed; if the value of the Boolean formula is false, the simple statement is ignored. The next statement in sequence following the if statement is then executed.

B. Ifeither Statements

1. Construction

ifeither statement: $\boxed{\text{ifeither clause}}$ $\boxed{\text{simple statement}}$ $\Big\{$ $\boxed{\text{orif clause}}$ $\boxed{\text{simple statement}}$ $\Big\}$ $\Big\{$ $\boxed{\text{orif clause}}$ $\boxed{\text{simple statement}}$ $\Big\}$ END#

ifeither clause: IFEITH# $\boxed{\text{Boolean formula}}$ \$

orif clause: $\Big\{$ $\boxed{\text{name}}$. $\Big\}$ ORIF# $\boxed{\text{Boolean formula}}$ \$

2. Meaning

The ifeither statement serves to specify the conditional execution of one of several simple statements.

The effect of an ifeither statement may be defined in terms of simpler statements. The statement

```
    IFEITH b0$ s0
n1. ORIF b1$ s1
n2. ORIF b2$ s2
    ........ END
```

where b0 , b1 , and b2 are Boolean formulas, s0 , s1 , and s2 are simple statements, and n1 and n2 are statement names, has the effect of being replaced during execution by

```
    IF b0$ BEGIN s0 GOTO nn$ END
n1. IF b1$ BEGIN s1 GOTO nn$ END
n2. IF b2$ BEGIN s2 GOTO nn$ END
    ..........
nn.
```

where nn is the (possibly implicit) name of the statement following the ifeither statement.

C. <u>For Statements</u>

1. <u>Construction</u>

for statement: $\boxed{\text{one-factor for statement}}$ |

$\boxed{\text{two-factor for statement}}$ | $\boxed{\text{three-factor for statement}}$

one-factor for statement: $\boxed{\text{one-factor for clause}}$

$\left| \boxed{\text{one-factor for clause}} \right|$ $\boxed{\text{simple statement}}$

two-factor for statement: $\left| \boxed{\text{incomplete for clause}} \right|$

$\boxed{\text{two-factor for clause}}$ $\left| \boxed{\text{incomplete for clause}} \right|$

$\boxed{\text{subordinate statement}}$

three-factor for statement: $\left| \boxed{\text{one-factor for clause}} \right|$

$\boxed{\text{complete for clause}}$ $\left| \boxed{\text{incomplete for clause}} \right|$

$\boxed{\text{subordinate statement}}$

incomplete for clause: $\boxed{\text{one-factor for clause}}$ |

$\boxed{\text{two-factor for clause}}$

complete for clause: $\boxed{\text{three-factor for clause}}$ | $\boxed{\text{all clause}}$

one-factor for clause: $\left| \boxed{\text{name}} \; . \right|$ FOR# $\boxed{\text{index}}$ =

$\boxed{\text{initial value}}$ \$

two-factor for clause: $\left| \boxed{\text{name}} \; . \right|$ FOR# $\boxed{\text{index}}$ =

$\boxed{\text{initial value}}$, $\boxed{\text{step value}}$ \$

three-factor for clause: $\left| \boxed{\text{name}} \; . \right|$ FOR# $\boxed{\text{index}}$ =

$\boxed{\text{initial value}}$, $\boxed{\text{step value}}$, $\boxed{\text{terminal value}}$ \$

all clause: $\left| \boxed{\text{name}} \; . \right|$ FOR# $\boxed{\text{index}}$ = ALL

($\boxed{\text{table designation}}$) \$

initial value: $\boxed{\text{numeric formula}}$

step value: $\boxed{\text{numeric formula}}$

terminal value: $\boxed{\text{numeric formula}}$

subordinate statement: $\boxed{\text{simple statement}}$ | $\boxed{\text{for compound}}$

for compound: BEGIN# $\boxed{\text{sentence list}}$ $\left|\ \boxed{\text{name}}\ .\ \right|$

$\boxed{\text{if clause}}$ END#

2. Meaning

A for statement serves to specify the iterative execution of a simple statement.

The effect of a for statement may be defined in terms of simpler statements. In the following discussion, n0 , n1 , n2 , etc., are statement names; v0 , v1 , v2 , etc., are indexes; f0 , f1 , f2 , etc., are initial values (numeric formulas); i0 , i1 , i2 , etc., are step values (numeric formulas); t0 , t1 , t2 , etc., are terminal values (numeric formulas); s is a simple statement; and ss is a subordinate statement.

The one-factor for statement

```
    FOR  v0 = f0$
n1. FOR  v1 = f1$
n2. FOR  v2 = f2$
```

has the effect of being replaced during execution by

```
    v0 = f0$
n1. v1 = f1$
n2. v2 = f2$
    . . . . .
    s
```

In other words, the indexes are assigned initial values in the order of their appearance in for clauses, and the simple statement executed once.

The two-factor for statement

```
         ........
         FOR  v0 = f0, i0$
    n1.  FOR  v1 = f1$
    n2.  FOR  v2 = f2, i2$
    n3.  FOR  v3 = f3$
         ........
         ss
```

has the effect of being replaced during execution by

```
         ........
         v0 = f0$
    n1.  v1 = f1$
    n2.  v2 = f2$
    n3.  v3 = f3$
         ........
    nm.  ss
         ........
         v2 = v2 + i2$
         v0 = v0 + i0$
         ........
         GOTO  nm$
```

where nm is the (possibly implicit) name of the subordinate statement.

In other words, the indexes are assigned initial values in the order of their appearance in for clauses; the subordinate statement is executed; indexes appearing in two-factor for clauses are increased by their step values (possibly negative) in reverse order of their appearance in for clauses; and the latter two operations are performed iteratively.

The three-factor for statement

```
         ........
         FOR  v0 = f0$
    n1.  FOR  v1 = f1, i1, t1$
    n2.  FOR  v2 = f2$
    n3.  FOR  v3 = f3, i3$
         ........
         ss
```

has the effect of being replaced during execution by

```
     . . . . . . . .
     v0 = f0$
n1.  v1 = f1$
n2.  v2 = f2$
n3.  v3 = f3$
     . . . . . . . .
nm.  IF i1 GR 0 AND v1 GR t1
     OR i1 LS 0 AND v1 LS t1$
     GOTO nn$
     ss
     . . . . . . . .
     v3 = v3 + i3$
     v1 = v1 + i1$
     . . . . . . . .
     GOTO nm$
nn.
```

where nm is an implicit statement name and nn is the (possibly implicit) name of the statement following the for statement.

In other words, the indexes are assigned initial values in the order of their appearance in for clauses; the index of the three-factor for clause is examined to see if it has exceeded the current terminal value (exceeding meaning in the direction of the current sign of the step value); the subordinate statement is executed; indexes appearing in two- and three-factor for clauses are increased by their step values (possibly negative) in reverse order of their appearance in for clauses; and the latter three operations are performed iteratively until the above test is met. If the test is met before the first execution of the subordinate statement, the statement is not executed at all.

The all clause

```
FOR v = ALL(a)$
```

where a is a table designation and v is an index, has the effect of being replaced during execution by

```
FOR v = 0, 1, NENT(a) - 1$
```

Thus, the all clause may be considered a special form of a three-factor for clause.

The for compound

```
BEGIN sl nl. IF bl$ END
```

where sl is a sentence list, has the effect of being replaced during execution by

```
BEGIN
        sl
   nl. IF NOT (bl)$ GOTO nn$
END
```

where nn is the (possibly implicit) name of the statement following the for statement.

The programmer should note that maximum optimization of two- and three-factor clauses is obtained only if the step and terminal values are both numeric formulas containing only numeric constants.

More than one three-factor for clause may actually appear in a three-factor for statement, but those other than the first are considered to be stripped of their terminal value and, in effect, to be two-factor for clauses.

All statement names in a for statement, except those prefixed to the first for clause, are considered to be within the for statement as far as their scope is concerned (see subsection VIII.B, Scope of Definitions).

D. Test Statements

1. Construction

test statement: TEST $ | TEST# $\boxed{\text{index}}$ $

2. Meaning

A test statement serves to specify an interruption in the normal sequence of operations within a for statement. The test statement must only appear within a two- or three-factor for statement.

The effect of a test statement which does not designate an index is that of a goto statement whose successor is the first implicit statement (executed by the innermost for statement containing the test statement) that increments an index.

The effect of a test statement which does designate an index is that of a goto statement whose successor is the implicit statement (executed by a for statement containing the test statement) that increments that index; if the index was defined in a one-factor for clause, the successor is the implicit statement which follows the point at which an implicit incrementing statement would have appeared if that index were defined in a two- or three-factor for clause.

VI. DATA DECLARATIONS

A. Item Descriptions

1. Construction

item description: $\boxed{\text{numeric item description}}$ |

$\boxed{\text{Boolean item description}}$ | $\boxed{\text{literal item description}}$ |

$\boxed{\text{status item description}}$

numeric item description: $\boxed{\text{integer item description}}$ |

$\boxed{\text{fixed item description}}$ | $\boxed{\text{floating item description}}$

integer item description: I# $\boxed{\text{size}}$ # $\boxed{\text{signing}}$ # $\boxed{\text{rounding}}$ #

$\boxed{\text{range}}$ | A# $\boxed{\text{size}}$ # $\boxed{\text{signing}}$ # $\boxed{\text{rounding}}$ # $\boxed{\text{range}}$

fixed item description: A# $\boxed{\text{size}}$ # $\boxed{\text{signing}}$ # $\boxed{\text{precision}}$ #

$\boxed{\text{rounding}}$ # $\boxed{\text{range}}$

floating item description: F# $\boxed{\text{rounding}}$ # $\boxed{\text{range}}$

Boolean item description: B

literal item description: $\boxed{\text{Hollerith item description}}$ |

$\boxed{\text{transmission item description}}$

Hollerith item description: H# $\boxed{\text{size}}$

transmission item description: T# $\boxed{\text{size}}$

status item description: S# $\boxed{\text{size}}$ # $\boxed{\text{status constant list}}$ |

S# $\boxed{\text{status constant list}}$

status constant list: $\boxed{\text{status constant}}$ | # $\boxed{\text{status constant}}$ |

size: $\boxed{\text{number}}$

signing: S | U

rounding: R | #

range: $\boxed{\text{numeric constant}}$ . . . $\boxed{\text{numeric constant}}$ | #

2. <u>Meaning</u>

Item descriptions are a part of data declarations which specify various characteristics of the quantities declared.

The integer item description serves to define a quantity which may contain only integer values. The size specifies the number of bits of which the quantity is composed, from 1 to 71 (if unsigned) or from 1 to 72 (if signed). The signing, if an S , specifies that the high-order bit is to represent the sign of the quantity; if a U , all bits are used for the magnitude and the quantity is considered an unsigned (i. e., implicitly positive) quantity. The rounding, if an R , specifies that any value assigned to it is to be rounded if the precision of the value exceeds that of the quantity; if no R is present, such a value will be truncated. The range, if present, specifies the minimum and maximum magnitude of any value which may be assigned to the quantity; however, the range is not used by the compiler.

The fixed item description serves to define a quantity which may contain only fixed values. The size, signing, rounding, and range are as noted above for integer quantities. The precision specifies the position of the binary point relative to the low-order bit of the quantity; if the precision is positive, it specifies the number of positions to the left, and if negative, the number to the right.

The floating item description serves to define a quantity which may contain only floating values. Rounding and range are as noted for integer quantities. The implied size is one word.

The Boolean item description serves to define a quantity which may contain only Boolean values. The implied size is one bit.

The Hollerith item description serves to define a quantity which may contain only Hollerith values. The size specifies the number of six-bit bytes of which the quantity is composed.

The transmission item description serves to define a quantity which may contain only transmission values. The size specifies the number of six-bit bytes of which the quantity is composed.

The status item description serves to define a quantity which may contain only status values. The status constant list specifies some or all of the values of the quantity. The first status constant written will represent the unsigned integer value 0; the second, 1; the third, 2; and so forth. The size, if given, specifies the number of bits of which the quantity is composed; in this case, the number of status constants written must not exceed 2^n, where n is the size given. If the size is not given, it will be determined as n by $2^{n-1} < m \leq 2^n$, where m is the number of written status constants. The size of a status quantity must not exceed 71 bits.

B. Simple Item Declarations

1. Construction

simple item description: ITEM# $\boxed{\text{name}}$ #

$\boxed{\text{item description}}$ \$ | ITEM# $\boxed{\text{name}}$ #

$\boxed{\text{item description}}$ #P# $\boxed{\text{preset value}}$ \$ |

ITEM# $\boxed{\text{name}}$ # $\boxed{\text{preset value}}$ \$

preset value: $\boxed{\text{signed numeric constant}}$ | $\boxed{\text{Boolean constant}}$ |

$\boxed{\text{literal constant}}$ | $\boxed{\text{status constant}}$

signed numeric constant: + $\boxed{\text{numeric constant}}$ |

- $\boxed{\text{numeric constant}}$ | $\boxed{\text{numeric constant}}$

2. Meaning

The simple item declaration serves to define a quantity as a simple item with the given name and characteristics.

The preset value specifies the initial value of the quantity. The effect of specifying a preset value is that of an assignment statement

$$v = c\$$$

written immediately after the declaration, where v is the simple item name and c is the preset value.

If no item description is given, the quantity is specified to be of the same type as the preset value, and of the size required to hold the value. The quantity is signed if a + or - is shown and unsigned otherwise; is unrounded; and if fixed, has a precision equal to that of the preset value. Also, the preset value must not be a status constant nor a Boolean 0 or 1 if no item description is given.

A simple item declaration must precede the first reference to the simple item.

C. Mode Declarations

1. Construction

mode declaration: MODE# $\boxed{\text{item description}}$ \$ | MODE# $\boxed{\text{preset value}}$ \$ | MODE# $\boxed{\text{item description}}$ #P# $\boxed{\text{preset value}}$ \$

2. Meaning

The mode declaration serves to define otherwise undefined quantities as simple items with given characteristics.

A mode declaration defines a name which appears in a main program (a program excluding its procedures) used as a simple item name only if:

a. It has not been declared previously in the main program in a simple item, array, table, file, or procedure declaration;

b. It is not defined in the COMPOOL as a simple item, array, table, or table item;

c. The first appearance of the name follows the appearance of this mode declaration; and

d. No other mode declaration in the main program or in any procedure appears between this mode declaration and the first appearance of the name.

A mode declaration defines a name which appears in a procedure used as a simple item name only if:

a. It has not been declared previously in the main program or in this procedure in a simple item, array, table, file, mode, or procedure declaration;

b. It is not defined in the COMPOOL as a simple item, array, table, or table item;

c. The first appearance of the name follows the appearance of this mode declaration; and

d. No other mode declaration in the main program or in any procedure appears between this mode declaration and the first appearance of the name.

When a name appears in the above-described context (except that no mode declaration precedes its first appearance), an implicit mode declaration of the form

```
MODE I 36 S$
```

is assumed to exist as the first sentence in the program.

Preset values are treated as specified for simple item declarations.

D. Array Declarations

1. Construction

array declaration: ARRAY# $\boxed{\text{name}}$ # $\boxed{\text{dimension list}}$ #

$\boxed{\text{item description}}$ \$ | ARRAY# $\boxed{\text{name}}$ #

$\boxed{\text{dimension list}}$ # $\boxed{\text{item description}}$ \$ $\boxed{\text{preset description}}$

dimension list: $\boxed{\text{dimension}}$ | # $\boxed{\text{dimension}}$ |

dimension: $\boxed{\text{number}}$

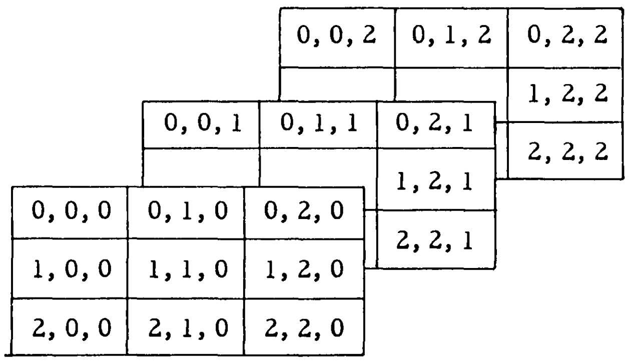
preset description: BEGIN# $\boxed{\text{preset list}}$ END#

preset list: | $\boxed{\text{preset description}}$ # | | | $\boxed{\text{preset value}}$ # |

2. Meaning

The array declaration serves to define an n-dimensional arrangement of quantities, each quantity with the given name and characteristics.

The dimensionality of the array is specified by the dimension list. The number of dimensions in the array is the number of dimensions written. The size of each dimension is given as the number written for each dimension. This number must be nonzero. Thus, the array is rectangular. The arrangement of quantities in the array can be illustrated by the following example.



where the list of numbers in a cell (i,j,k) represents the values of subscripts in the subscript list of a variable naming this three-dimensional array. The row number is indicated by i , the column number by j , and the matrix number by k . Each of the dimensions has a size of three. Note that the first element in a dimension is numbered zero. The illustration may be extended in the obvious manner for any other array. Any number and size of dimensions (except zero) is permissible.

Arrays are stored column by column in internal storage, e.g., (0, 0, 0), (1, 0, 0), (2, 0, 0), (0, 1, 0), (1, 1, 0), . . . , (2, 2, 0), (0, 0, 1), (1, 0, 1), . . . , (1, 2, 2), (2, 2, 2).

The preset description, if present, specifies initial values for quantities in the array. To determine the correspondence between individual quantities and preset values, consider the following example for the three-dimensional array illustrated above:

BEGIN	BEGIN	BEGIN	0,0,0	0,1,0	0,2,0	END		
		BEGIN	1,0,0	1,1,0	1,2,0	END		
		BEGIN	2,0,0	2,1,0	2,2,0	END	END	
	BEGIN	BEGIN	0,0,1	0,1,1	0,2,1	END		
		BEGIN	1,0,1	1,1,1	1,2,1	END		
		BEGIN	2,0,1	2,1,1	2,2,1	END	END	
	BEGIN	BEGIN	0,0,2	0,1,2	0,2,2	END		
		BEGIN	1,0,2	1,1,2	1,2,2	END		
		BEGIN	2,0,2	2,1,2	2,2,2	END	END	END

In this example the preset values have been replaced by the subscript list for the corresponding quantity. Written preset values are paired with the quantities indicated from left to right within innermost BEGIN END brackets, and each preset value is assigned to its paired quantity in the manner specified for simple items. Not all quantities need to have paired values; values may be missing from the right within innermost BEGIN END brackets. However, all brackets necessary to indicate the dimensions of the array must be present.

The correspondence illustrated above may be extended for arrays with more than three dimensions, and for arrays with two dimensions. For one-dimensional arrays, the single BEGIN END brackets enclose all preset values even though in this case they each correspond to elements of different rows.

The array declaration must precede the first reference to the array.

E. Independent Overlay Declarations

1. Construction

independent overlay declaration: OVERLAY#

$\boxed{\text{independent data sequence list}}$ \$

independent data sequence list: $\boxed{\text{independent data sequence}}$

$\Big| = \boxed{\text{independent data sequence}} \Big|$

independent data sequence: $\boxed{\text{independent data}}$

$\Big| , \boxed{\text{independent data}} \Big|$

independent data: $\boxed{\text{simple item name}} \;\Big|\; \boxed{\text{array name}} \;\Big|$

$\boxed{\text{table name}}$

2. Meaning

The independent overlay declaration serves to specify sequences in which simple items, arrays, and tables are to be arranged in internal storage; to specify overlaying of such sequences.

Within an independent data sequence, independent data are assigned to sequential addresses in the order written. Within an independent data sequence list, each independent data sequence begins at the same address.

A name must not appear more than once in a given independent overlay declaration, but may appear in more than one such declaration if the effect is logically consistent. For overlay purposes, a simple item begins with the first word containing it, an array begins with the first word containing it, and a table begins with its nent word. The meaning of values assigned to overlaid data (whether by assignment statements, preset values, etc.) may be determined only by the programmer, and it is his responsibility to avoid inconsistencies.

F. Ordinary Table Declarations

1. Construction

ordinary table declaration: $\text{TABLE\#}\ \boxed{\text{naming}}\ \#\ \boxed{\text{length}}\ \#$

$\boxed{\text{structure}}\ \#\ \boxed{\text{packing}}\ \$\ \boxed{\text{ordinary entry description}}$

naming: $\boxed{\text{name}}\ |\ \#$

length: $\text{V\#}\ \boxed{\text{number}}\ |\ \text{R\#}\ \boxed{\text{number}}$

structure: S | P | #

packing: D | M | N | #

ordinary entry description: $\text{BEGIN\#}\ \boxed{\text{ordinary entry declaration}}$

$|\ \boxed{\text{ordinary entry declaration}}\ |\ \text{END\#}$

ordinary entry declaration: $\boxed{\text{ordinary table item declaration}}\ |$

$\boxed{\text{dependent overlay declaration}}$

ordinary table item declaration: $\text{ITEM\#}\ \boxed{\text{name}}\ \#$

$\boxed{\text{item description}}\ \$\ |\ \text{ITEM\#}\ \boxed{\text{name}}\ \#\ \boxed{\text{item description}}\ \$$

$\text{BEGIN\#}\ |\ \boxed{\text{preset value}}\ \#\ |\ \text{END\#}$

dependent overlay declaration: OVERLAY#

$\boxed{\text{dependent data sequence list}}\ \$$

dependent data sequence list: $\boxed{\text{dependent data sequence}}$

$|\ =\ \boxed{\text{dependent data sequence}}\ |$

dependent data sequence: $\boxed{\text{table item name}}\ |\ ,\ \boxed{\text{table item name}}\ |$

2. Meaning

The ordinary table declaration serves to define a two-dimensional arrangement of quantities. In a serial table, all quantities in a column are defined as a single entry and all quantities in a row are defined as instances of a table item. In a parallel table, the reverse is true.

Thus, a table item is a vector of quantities each with the same name and characteristics, and an entry is composed of one instance of each item.

An ordinary table declaration defines a table in which the compiler, not the programmer, specifies the exact positioning of each quantity within an entry. The table may or may not be named.

The table may be of variable or rigid length, the former indicated by V and the latter by R . A rigid table has a fixed number of entries given by the number following the R . A variable table has a variable number of entries, with the maximum number of entries given by the number following the V . Both kinds of tables are prefixed in storage with a word which in a rigid table yields a value equal to the fixed number of entries, and which in a variable table yields or is assigned a value equal to the current number of entries. This word is the nent word.

The table is structured in serial or in parallel, the former indicated by S and the latter by P . If no structure is specified, parallel is assumed. Tables, like arrays, are stored "column by column"; thus, serial tables are stored "entry after entry" and parallel tables are stored "all instances of a table item after all instances of a table item." (An exception to the latter is that each word will be stored as a column for multiword items.) Variable tables must be structured in serial.

The table items making up an entry may or may not be "packed." If an N or no packing is specified, each table item is stored beginning in a different word. If an M or D is specified, the table items are stored in order to minimize unused bits. If further knowledge of the exact allocation of table items within an entry is required, the programmer is advised to use the specified table declaration.

The ordinary entry description specifies the composition of an entry in terms of its table items, via ordinary entry declarations.

The ordinary table item declaration defines a table item as a set of quantities each with given name and characteristics. The preset values, if given, specify initial values for these quantities. The first value is assigned to the first instance of the table item; the second value, to the second instance; and so forth, in the manner specified for simple items. Not all instances need corresponding preset values.

The dependent overlay declaration specifies sequences in which the table items are arranged within an entry, and specifies overlaying of such sequences. If N or no packing is specified for the table, the table items appearing in a given dependent data sequence are arranged in the order written; and within a given dependent data sequence list, each dependent data sequence begins at the same address for a given entry. If M or D packing is specified for the table, the effect is to begin dependent data sequences within a given dependent data sequence list at the same address for a given entry; the order within a dependent data sequence is not defined.

A table item name must not appear more than once in a given dependent overlay declaration, but may appear in more than one such declaration if the effect is logically consistent. A table item must appear in an ordinary table item declaration before it may appear in a dependent overlay declaration.

The meaning of values assigned to overlaid table items (whether by assignment statements, preset values, etc.) may be determined only by the programmer, and it is his responsibility to avoid logical inconsistencies.

The table declaration must precede the first reference to the table or table items.

G. Specified Table Declarations

1. Construction

specified table declaration: TABLE# $\boxed{\text{naming}}$ # $\boxed{\text{length}}$ #

$\boxed{\text{structure}}$ # $\boxed{\text{width}}$ \$ $\boxed{\text{specified entry description}}$

width: $\boxed{\text{number}}$

specified entry description: BEGIN#

$\boxed{\text{specified table item declaration}}$

$\left| \boxed{\text{specified table item declaration}} \right|$ END#

specified table item declaration: ITEM# $\boxed{\text{name}}$ #

$\boxed{\text{item description}}$ # $\boxed{\text{position}}$ \$ | ITEM# $\boxed{\text{name}}$ #

$\boxed{\text{item description}}$ # $\boxed{\text{position}}$ \$ BEGIN#

$\left| \boxed{\text{preset value}} \ \# \right|$ END#

position: $\boxed{\text{word}}$ # $\boxed{\text{bit}}$ # $\boxed{\text{packing}}$

word: $\boxed{\text{number}}$

bit: $\boxed{\text{number}}$

2. Meaning

The specified table declaration serves to define a two-dimensional arrangement of quantities similar to that defined by the ordinary table declaration. In the specified table declaration the programmer, not the compiler, specifies the exact positioning of each quantity within an entry.

The naming, length, and structure of a specified table are as noted for ordinary tables in the preceding subsection. The width specifies the number of words in an entry; this is the actual width of an entry regardless of the positioning of individual table items.

The specified entry description specifies the composition of an entry in terms of its table items and gives the exact position of each table item within the entry. Dependent overlay declarations are not a part of the specified entry description, since the programmer may accomplish such positioning explicitly.

The specified table item declaration defines a table item as a set of quantities, each with given name and characteristics. The preset values are specified as noted for ordinary table item declarations in the preceding subsection.

The position given in the specified table item declaration specifies the word of the entry and the bit of the word in which the table item begins. The first word of an entry is indicated by 0; the second, by 1; and so forth. The first bit of a word is indicated by 0; the second, by 1; and so forth. (Table items with a size less than or equal to 36 bits must not be positioned in more than one word; and those with a size greater than 36 bits, but less than or equal to 72 bits, must not be positioned in more than two words, etc.) If the packing is given, it is not used by the compiler but may be used by the programmer to describe the situation resulting from his positioning of the item.

If the table is of serial structure, it is permissible to position table items beyond the end of the entry (as specified by the width). This will cause the position of such a table item to extend into the next entry, thereby overlaying the initial table items of the next entry. Such a table item, however, is referenced as if it were a part of the current entry.

H. Like Table Declarations

1. Construction

like table declaration: TABLE# $\boxed{\text{name}}$ # $\boxed{\text{length}}$ # $\boxed{\text{structure}}$ # $\boxed{\text{packing}}$ # L \$ | TABLE# $\boxed{\text{name}}$ # $\boxed{\text{structure}}$ # $\boxed{\text{packing}}$ # L \$

2. Meaning

The like table declaration serves to define a table which is like a previously declared table (ordinary or specified). The previously declared table (the pattern table) may be declared at any point earlier in the program where tables may be declared.

The name of the like table must be identical to the name of the pattern table except for a suffixed letter or numeral. The table defined by the like table declaration is then identical in composition to the pattern table except that any length, structure, or packing specified in the like table declaration overrides that given in the pattern table declaration, and except that no preset values are assigned to the like table.

The names of the table items in the like table are identical to the names of the table items in the pattern table except that the letter or numeral noted above is suffixed to each table item name.

VII. PROCESS DECLARATIONS

A. Switch Declarations

1. Construction

index switch declaration: SWITCH# $\boxed{\text{name}}$ = ($\boxed{\text{index switch list}}$) \$

index switch list: $\boxed{\text{index switch element}}$ | , $\boxed{\text{index switch element}}$ |

index switch element: $\boxed{\text{sequence designation}}$ | #

item switch declaration: SWITCH# $\boxed{\text{name}}$ ($\boxed{\text{switch item}}$) =

($\boxed{\text{item switch list}}$) \$

switch item: $\boxed{\text{simple item name}}$ | $\boxed{\text{array name}}$ | $\boxed{\text{table item name}}$

item switch list: $\boxed{\text{item switch element}}$ | , $\boxed{\text{item switch element}}$ |

item switch element: $\boxed{\text{preset value}}$ = $\boxed{\text{sequence designation}}$

file switch declaration: see subsection IX. A, File Declarations and Status

2. Meaning

A switch declaration serves to define a set of operations which may be invoked from various points in the program by goto statements. The effect of an index switch declaration may be defined in terms of simpler statements. The declaration

```
SWITCH x = (,e1,,e2,e3,)$
```

where x is an index switch name and e1 , e2 , and e3 are sequence designations, when invoked by the statement

```
GOTO x($n$)$
```

where n is a numeric formula, has the effect of replacing the goto statement during execution by the compound statement

```
BEGIN
      IF  f(n)  EQ  0$  GOTO  e$
      IF  f(n)  EQ  1$  GOTO  e1$
      IF  f(n)  EQ  2$  GOTO  e$
      IF  f(n)  EQ  3$  GOTO  e2$
      IF  f(n)  EQ  4$  GOTO  e3$
      IF  f(n)  EQ  5$  GOTO  e$
 END
```

where f is a procedure name that designates an implicit function yielding the integer value of the actual parameter (by truncation if necessary), and e is the (possibly implicit) name of the statement following the original goto statement.

In other words, the value of the numeric formula selects one of the switch elements according to the position of the element on the list. A value of 0 selects the first element; 1, the second; and so forth. When an element is missing, the goto statement has no effect. The value must be one which corresponds to a position in the list.

The effect of an item switch declaration can be defined in terms of simpler statements. The declaration

```
SWITCH  x(y)  = (v0 = e0, v1 = e1, v2 = e2)$
```

where x is an item switch name; y is a simple item name; v0 , v1 , and v2 are preset values; and e0 , e1 , and e2 are sequence designations, when invoked by the statement

```
GOTO  x$
```

has the effect of replacing the goto statement during execution by the compound statement

```
BEGIN
    IF y EQ v0$ GOTO e0$
    IF y EQ v1$ GOTO e1$
    IF y EQ v2$ GOTO e2$
 END
```

If y is an array name or a table item name and the switch is invoked by the statement

```
GOTO x($n$)$
```

where n is a subscript (for a table item) or a subscript list (for an array), the replacement is

```
BEGIN
    IF y($n$) EQ v0$ GOTO e0$
    IF y($n$) EQ v1$ GOTO e1$
    IF y($n$) EQ v2$ GOTO e2$
 END
```

In other words, the value of the quantity y selects one of the sequence designators according to a match with the corresponding preset value. If the value of the quantity does not match any of the preset values, the goto statement has no effect.

In either the item or index switch declaration, sequence designations which reference other switch designations are understood to cause further effective replacements as required.

A switch declaration defined within a for statement must not be invoked by a goto statement outside that for statement.

B. Close Declarations

1. Construction

close declaration: CLOSE# $\boxed{\text{name}}$ $ BEGIN# $\boxed{\text{sentence list}}$

END#

2. Meaning

A close declaration serves to define a set of operations that may be invoked from various points in the program by goto statements. The effect of a close declaration, when invoked by a goto statement, is to replace the goto statement during execution by the sentence list given in the close declaration, enclosed in its BEGIN END brackets.

Close declarations must be placed in a program so that they may only be reached by the invoking goto statements. No close declaration may contain a statement which directly or indirectly invokes it (i. e., no recursion).

A close declaration defined within a for statement must not be invoked by a goto statement outside that for statement, nor by a goto statement in a procedure not containing that close declaration.

C. Return Statements

1. Construction

return statement: RETURN $

2. Meaning

The return statement must only appear within the sentence list of a close declaration or a procedure declaration. The effect of a return statement is to designate the statement following the invoking goto statement (for a close declaration) as the next statement to be executed, or to designate a particular implicit statement (for a procedure declaration) as the next statement to be executed (see the next subsection).

D. Procedure Declarations

1. Construction

procedure declaration: $\boxed{\text{procedure heading}}$ $\boxed{\text{declaration list}}$ $\boxed{\text{procedure body}}$

procedure heading: PROC# $\boxed{\text{name}}$ ($\boxed{\text{formal parameter list}}$) \$ | PROC# $\boxed{\text{name}}$ \$

formal parameter list: $\boxed{\text{formal input parameter list}}$ $\boxed{\text{formal output parameter list}}$

formal input parameter list: $\boxed{\text{formal input parameter}}$ $\left| \, , \boxed{\text{formal input parameter}} \right|$ | #

formal output parameter list: = $\boxed{\text{formal output parameter}}$ $\left| \, , \boxed{\text{formal output parameter}} \right|$ | #

formal input parameter: $\boxed{\text{simple item name}}$ | $\boxed{\text{array name}}$ | $\boxed{\text{table name}}$ | $\boxed{\text{close name}}$.

formal output parameter: $\boxed{\text{simple item name}}$ | $\boxed{\text{array name}}$ | $\boxed{\text{table name}}$ | $\boxed{\text{statement name}}$.

declaration list: $\left| \boxed{\text{declaration}} \right|$

procedure body: BEGIN# $\boxed{\text{sentence list}}$ END#

2. Meaning

A procedure declaration serves to define a set of operations that may be invoked from various points in a program by procedure statements or functions.

The effect of a procedure declaration can be defined in terms of simpler statements. The declaration

```
PROC  p(f1, f2, f3, f4. = f5, f6, f7, f8.)$ d
BEGIN  s  END
```

where p is a procedure name; f1 and f5 are simple item names; f2 and f6 are array names; f3 and f7 are table names; f4 is a close name; f8 is a statement name; d is a declaration list including declarations for f1 , f2 , f3 , f5 , f6 , and f7 ; and s is a sentence list

when invoked by the procedure statement

```
p(a1, a2, a3, a4. = a5, a6, a7, a8.)$
```

where a1 is a formula, a5 is a variable, a2 and a6 are array names, a3 and a7 are table names, a4 is a close name, and a8 is a statement name

has the effect of replacing the procedure statement during execution by the compound statement

```
BEGIN
        d
        f1 = a1$
        OVERLAY  f2 = a2$
        OVERLAY  f3 = a3$
        OVERLAY  f6 = a6$
        OVERLAY  f7 = a7$
        s
        GOTO  x$
        CLOSE  f4$  BEGIN  GOTO  a4$  END
  f8.   a5 = f5$
        GOTO  a8$
  x.    a5 = f5$
END
```

where x is the implicit name of the statement which any return statement in s is understood to reference. Any subscript list in a5 is understood to be evaluated before such replacement takes effect.

In other words, the following operations are performed when a procedure declaration is invoked by a procedure statement:

a. Simple item formal input parameters are assigned the values of corresponding formula actual input parameters.

b. Array and table formal parameters are considered to be overlaid with corresponding array and table actual parameters.

c. Close and statement name formal parameters are considered to be identical with corresponding close and statement name actual parameters.

d. The subscripts (if any) of variable actual output parameters are evaluated.

e. The procedure body is executed.

f. Variable actual output parameters are assigned the values of corresponding simple item formal output parameters. This operation is performed only if the procedure body is exited via a return statement, by normal statement sequencing, or via a formal output parameter that is a statement name.

Thus, each formal parameter must correspond to an actual parameter (and vice versa) by their position in the parameter lists as follows:

a. Formal input simple items to actual input formulas.

b. Formal output simple items to actual output variables.

c. Formal tables to actual tables.

d. Formal arrays to actual arrays.

e. Formal close names to actual close names.

f. Formal statement names to actual statement names.

In addition, the declaration list must contain declarations for all formal simple item, array, and table names. The declaration list and procedure body of a procedure declaration must not contain any procedure declaration. The procedure body must not contain a statement which directly or indirectly invokes this procedure declaration (i. e., no recursion).

The effect of a procedure declaration when invoked by a function is similar to when it is invoked by a procedure statement, except that the declaration list must include a simple item declaration for the procedure name, and the simple item must be assigned a value within the procedure; that value is then the value of the function that invoked the procedure declaration.

Procedure declarations must be placed in a program such that they may be reached only by the invoking procedure statement or function.

VIII. PROGRAMS

A. Programs

1. Construction

program: START# $\boxed{\text{sentence list}}$ TERM# $\boxed{\text{statement name}}$ \$ |

START# $\boxed{\text{sentence list}}$ TERM \$

2. Meaning

A program serves to define a set of operations that may be invoked by the operating system. If a statement name is given, it designates the first statement in the program to be executed. If a statement name is not given, the first statement to be executed is the first sequential statement in the program. In either case, the first statement executed is not one which is within any declaration.

Within restrictions given elsewhere in this text (i. e., under direct statements), the sign string comprising a program may be written with no regard for any particular line format, except that no \$ may appear in the first column of a line.

B. Scope of Definitions--Meaning

Names are characterized by the scope of their definition. Scope of definition refers to the part of the program for which the name is considered to be defined and hence may be referenced or invoked.

There are three basic kinds of scope--local, global, and system. A name with local scope is defined over the procedure in which it is declared. A name with global scope is defined over the main program (the program excluding its procedures) in which it is declared, as well as over any procedures in which it is not locally defined. A name with system scope (defined in the COMPOOL or library) is defined over all main programs in which it is not defined globally, as well as over any procedures in which it is neither globally nor locally defined.

The scope of a name is determined from the point in the program at which it is declared, not at any point at which it may be referenced or invoked.

Two identically constructed names are considered to be different names if they are declared in different categories. The two categories are goto names and set names (see subsection II.E, Names and Indexes). Hence, even if their scopes coincide, no confusion can arise. Otherwise, the scope of two or more identically constructed names must not coincide.

Exhibit 2 summarizes the methods by which names may be declared and the scope they are consequently given.

Indexes, like names, are also characterized by their scope of definition. The scope of an index includes the for statement which contains the for clause in which the index appears to the left of the = . This scope includes other for statements contained in that for statement but does not include procedure declarations contained in that for statement.

In particular, the scope of an index begins with the above-described appearance in a for clause and, thus, may be referenced in the step- and terminal-value formulas of that clause as well as in subsequent for clause formulas.

As with names, the scope of two or more identically constructed indexes must not coincide. By the very concept of scope, reference to a name or index from outside its scope is not defined.

EXHIBIT 2 - SCOPE OF NAMES

Name	Local Scope (defined in procedure by)	Global Scope (defined in main program by)	System Scope (defined in)
Statement name	Prefixing to a statement(1) or appearing as formal output parameter	Prefixing to a statement(1)	
Switch name	Switch declaration(1)	Switch declaration(1)	
Close name	Close declaration(1) or appearing as formal input parameter	Close declaration(1)	
Simple item name	Simple item declaration(2) or mode declaration(3)	Simple item declaration(2) or mode declaration(4)	COMPOOL
Array name	Array declaration(2)	Array declaration(2)	COMPOOL
Table name	Table declaration(2)	Table declaration(2)	COMPOOL
Table item name	Table declaration(2)	Table declaration(2)	COMPOOL
File name	File declaration(2)	File declaration(2)	
Procedure name		Procedure declaration	Library

Notes: (1) If defined within a for statement, the name must not be referenced from outside the innermost for statement containing the definition.

(2) Definition must precede the first reference to the name.

(3) If not defined in main program by earlier declaration, nor in COMPOOL, then defined at first appearance by last previous mode declaration.

(4) If not defined in main program or this procedure by earlier declaration (including mode), nor in COMPOOL, then defined at first appearance by last previous mode declaration.

C. Define Declarations

1. Construction

define declaration: DEFINE# $\boxed{\text{defined identifier}}$

" $\left| \boxed{\text{symbol}} \# \right|$ " \$

defined identifier: $\boxed{\text{identifier}}$

2. Meaning

Notwithstanding the remainder of the language, the define declaration serves to define an identifier that may stand in place of the string of symbols, given within the " brackets, anywhere in the program.

The string of symbols must not include a comment nor a " bracket. Any identifier (constructed identically to a defined identifier) that appears within a symbol (e. g., within a status constant or comment) is not considered to be that defined identifier. Wherever else a defined identifier appears in a program (within its scope) it is considered to be replaced by the string of symbols defined for it before the remainder of the program is examined.

The scope of a defined identifier extends from its define declaration to the next (if any) define declaration in which that defined identifier is again defined.

Thus, for instance, the appearance of the defined identifier, among the symbols of another define declaration written within its scope, leads to the replacement of the defined identifier at that appearance before that other define declaration is effected.

IX. FILE OPERATIONS

A. File Declarations and Status

1. Construction

file declaration: FILE# $\boxed{\text{name}}$ # $\left|\ \boxed{\text{status constant}}\ \#\ \right|$

$\boxed{\text{device code}}$ \$

logical file code: R $\boxed{\text{number}}$

file status relation: $\boxed{\text{file name}}$ # $\boxed{\text{relational operator}}$ #

$\boxed{\text{status formula}}$

file switch declaration: SWITCH# $\boxed{\text{name}}$ ($\boxed{\text{file name}}$) =

($\boxed{\text{file switch list}}$) \$

file switch list: $\boxed{\text{file switch element}}$ $\left|\ ,\ \boxed{\text{file switch element}}\ \right|$

file switch element: $\boxed{\text{status constant}}$ = $\boxed{\text{sequence designation}}$

2. Meaning

A file declaration serves to define (possibly a segment of) a given external storage device as a file in which values may be stored and/or from which values may be retrieved. There are currently six types of such devices--magnetic tape, drum, disc, card reader and punch, and printer. The logical file code specifies the particular device in which the file resides.

Associated with each logical file is a set of conditions, any of which may arise during operations involving these files. To permit reference to these conditions in a program, the file declaration may include a list of status constants, each of which designates a type of condition. The correspondence between status constants and conditions is made through the order in which the status constants are listed in the file declaration. In this context, the file name is treated similar to status items. The first status constant represents an unsigned integer value of 0; the second, 1; the third, 2; and so forth.

Logical file conditions may be tested for through the file status relation, which is regarded exactly as a status relation. The file switch declaration is regarded exactly as an item switch declaration, where the switch item is the file name again treated as a status item.

The file declaration must precede the first reference to the file.

B. In and Out Statements

1. Construction

in statement: IN ($\boxed{\text{function code}}$, $\boxed{\text{file name}}$) $

out statement: OUT ($\boxed{\text{function code}}$, $\boxed{\text{file name}}$) $

function code: $\boxed{\text{number}}$

2. Meaning

Values stored in files are organized within a file in segments, which are groups of records. A record is the set of values recorded as the result of the execution of a single ion statement.

The out statement specifies a file by name to which values will be or have been transmitted. It designates via the function code whether to open or to close the file.

The in statement specifies a file by name from which values will be or have been transmitted. It designates via the function code whether to open or to close the file.

A file which is currently open may send or receive values. A file which is currently closed is prohibited from sending or receiving values.

C. Ion Statements

1. Construction

ion statement: IO $\boxed{\text{number}}$ ($\boxed{\text{manipulation list}}$) \$ |

IO $\boxed{\text{number}}$ ($\boxed{\text{transmission list}}$) \$

manipulation list: $\boxed{\text{function code}}$, $\boxed{\text{file name}}$,

$\boxed{\text{terminal action}}$ | $\boxed{\text{function code}}$, $\boxed{\text{file name}}$

transmission list: $\boxed{\text{function code}}$, $\boxed{\text{file name}}$,

$\boxed{\text{terminal action}}$, $\boxed{\text{storage list}}$

storage list: $\boxed{\text{first word}}$, $\boxed{\text{number of words}}$ |

$\boxed{\text{first word}}$

terminal action: $\boxed{\text{procedure name}}$ | 0

first word: $\boxed{\text{simple item name}}$ | $\boxed{\text{array name}}$ | $\boxed{\text{table name}}$ |

$\boxed{\text{table name}}$ (\$ $\boxed{\text{subscript}}$ \$) | NENT ($\boxed{\text{table name}}$)

number of words: $\boxed{\text{numeric formula}}$

2. Meaning

Ion statements either manipulate external storage devices, or transmit values to or from files on those devices. The number following IO in an ion statement must not exceed four numerals and must be unique to each ion statement in a program.

If the ion statement is to designate device manipulation, it must include a manipulation list; if it is to designate value transmission, it must include a transmission list.

The terminal action, if a procedure name, must designate a parameterless procedure that will be invoked when the operation started by the ion statement reaches completion. The procedure will then be invoked as though a procedure statement naming that procedure were written at the point in the program reached when the operation is completed. If the terminal action is zero or absent, no procedure will be invoked.

The first word specifies the address in internal storage of the first value to be transmitted. The number of words specifies the number of words or entries to be transmitted, and must yield a nonnegative integer value. Only the following combinations are defined:

First Word	Number of Words	Words Transmitted	Entries Transmitted	NENT Word Transmitted	Used With Serial Structure	Used With Parallel Structure
Simple Item Name	n > 0	n				
Array Name	n > 0	n				
Table Name	Absent		Declared Number	No	Yes	Yes
Table Name	n > 0		n	No	Yes	No
Table Name ($subscript$)	n > 0		n	No	Yes	No
NENT (table name)	Absent or 0			Yes	Yes	Yes
NENT (table name)	n > 0		n	Yes	Yes	No

Function codes in ion statements are defined in the operating system.

D. Wait Statements

1. Construction

wait statement: WAIT (IO $\boxed{\text{number}}$) $

2. Meaning

The wait statement causes further sequencing of operations to be delayed until the operation begun by the ion statement with the same number has been completed.

X. DIRECT OPERATIONS

A. Direct Statements

1. Construction

direct statement: DIRECT# $\left\{ \boxed{\text{direct code}}\ \# \right\}$ JOVIAL#

direct code: $\boxed{\text{machine code}}\ \Big|\ \left\{ \boxed{\text{name}}\ .\ \right\}\ \boxed{\text{assign statement}}$

machine code: see R-731A, GE-635 JOVIAL Compiler Design Document (Supplement)

assign statement: see next subsection, Assign Statements

2. Meaning

The direct statement serves to define a set of operations that may be expressed in machine-oriented language. This language must consist entirely of certain legal GE-635 machine codes.

Each machine code is considered a symbol, and the entire line (or set of lines) on which it is written is considered part of the symbol. Thus, assign statements and DIRECT JOVIAL brackets must not appear on lines containing machine codes. Machine codes may address locations designated by names defined in JOVIAL. The interpretation of such is given in subsection II.E, Names and Indexes.

A label of a machine code, if also constructed as a legal machine code label and a legal JOVIAL statement name, becomes a defined statement name as though defined on a JOVIAL statement.

A statement name prefixed to the DIRECT bracket of a direct statement is considered to designate the first executable instruction within the DIRECT JOVIAL brackets.

Index registers used in direct statements should be saved on entrance and restored on exit from sequences of machine codes.

B. Assign Statements

1. Construction

assign statement: ASSIGN# $\boxed{\text{accumulator designation}}$ = $\boxed{\text{basic variable}}$ \$ | ASSIGN# $\boxed{\text{basic variable}}$ = $\boxed{\text{accumulator designation}}$ \$

accumulator designation: A ($\boxed{\text{precision}}$) | A ( )

2. Meaning

In the first construction the assign statement serves to assign the value of the basic variable to the machine accumulator; in the second construction it serves to assign the value held in the machine accumulator to the basic variable.

The precision, or its absence, specifies the characteristics of the machine accumulator (for that statement only) in the manner of a simple item, as follows:

Precision, p	Type of Basic Variable	Equivalent Item Description of Accumulator
$p \neq 0$	numeric	A 72 S p
$p = 0$	numeric	I 72 S
$p = 0$	Boolean	B
$p = 0$	Hollerith	H 12
$p = 0$	transmission	T 12
$p = 0$	status	S 71 . . .
p absent	numeric	F

The effect of the assign statement, so described, is equivalent to an assignment statement.

INDEX OF CONSTRUCTIONS

INDEX OF CONSTRUCTIONS
(Continued)

INDEX OF CONSTRUCTIONS
(Continued)

INDEX OF CONSTRUCTIONS
(Continued)

INDEX OF CONSTRUCTIONS
(Continued)

DOCUMENT REVIEW SHEET

TITLE:	GE-625/635 Jovial Compiler Reference Manual
CPB #:	XCPB-1201

Name: ______________________

Position: ______________________

Address: ______________________

____

CHECK ONE:

☐ Additional information would be helpful on following subjects.

☐ Errors indicated and pages where errors occur.

☐ Usefulness of manual could be improved as noted.

My comments are: ______________________

Please cut along this line

FOLD

FIRST CLASS
PERMIT, No. 4332
PHOENIX, ARIZONA

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY

GENERAL ELECTRIC COMPUTER DEPARTMENT
13430 NORTH BLACK CANYON HIGHWAY
PHOENIX, ARIZONA - 85001

ATTENTION: Program Documentation
Systems and Processors Operation

FOLD

Progress Is Our Most Important Product

GENERAL ELECTRIC

Computer Department • Phoenix, Arizona

LITHO IN U.S.A.

GENERAL ELECTRIC
COMPUTERS

GE-625/635 JOVIAL COMPILER

ADVANCE INFORMATION

GENERAL ELECTRIC

XCPB-1187 /

GE-625/635 JOVIAL COMPILER

SYSTEM DESCRIPTION

November 1965

GENERAL ELECTRIC

COMPUTER DEPARTMENT

PREFACE

This advance information manual is provided by General Electric Computer Department for the user and operator of the GE-625/635 JOVIAL compiler.

This manual describes the external characteristics and general internal characteristics of JOVIAL. Descriptive material falls into two general categories: (1) description of the compiler system environment -- of compiler inputs and outputs, interfaces, operating procedures, and restrictions; and (2) description of the compiler system's performance of its job -- of the methods used and the functional components of the compiler.

Comments concerning this publication should be addressed to Programming Documentation, General Electric Computer Department, P.O. Drawer 2961, Phoenix, Arizona, 85002.

TABLE OF CONTENTS

ILLUSTRATIONS

1. INTRODUCTION

The JOVIAL programming language was developed, beginning in early 1958, as a language suitable for military command and control applications. It is largely computer independent and is amenable to handling real-time problems as well as a wide variety of data forms. Based on ALGOL with extensions to match command and control objectives, JOVIAL has been officially adopted by both the Army and Navy as their standard programming language for command and control systems. The Air Force, although without "official" sanction, has made such extensive use of JOVIAL that it could be considered their "defacto" standard.

The GE-625/635 JOVIAL Compiler will process a carefully selected subset of the J3 version of JOVIAL. The compiler is syntax-controlled and operates according to the logic discussed in the following paragraphs. Several key items will first be defined.

The terms "syntax" and "syntax-controlled" are basic to this method of compilation. Syntax is the set of grammatical rules of construction of the JOVIAL language by which the entire logic of the compiler is built up; hence, the description, "syntax controlled." Parsing is the actual decoding of a JOVIAL statement and the discovery of its structure by using the JOVIAL syntax. The result of parsing a statement in the GE-625/635 JOVIAL compiler is a "Polish notation" representation of its basic components. Polish notation is merely a convenient logical format in which to hold the statement for input to the succeeding phases of the compiler.

Information that is generated or accumulated during compilation is kept in specialized tables called rolls. The most important characteristic of a roll is its ability to change size dynamically. During compilation, a roll continually changes size between fixed limits; if necessary, these fixed limits can also be changed dynamically. A synonym for a roll is "push-down, pop-up table." Since the roll is the heart of information storage in the compiler, POP's (program operators) are the basic tools of compiler processing logic. A POP is an instruction that, when interpreted, causes some compiler function to be performed. Although each individual POP function is small, the complete repertoire of POP's supplies all elementary compilation functions required to build the compiling logic.

The various phases of the compiler operate at four different levels of detail. From the largest to the smallest these are: the program, block, statement, and element levels. A program is an entire JOVIAL program. A block is a section of code between two transfer targets. A statement is a complete JOVIAL statement. An element is an independent subexpression within a JOVIAL statement.

The compiler consists of eight logical functions or sections -- Parse, Affirm, Influence, Diagnose, Finalize, Standardize, Generate, and Assemble -- plus a special Interlude phase. Each of these compiler sections is discussed in detail in Chapter 9; the overall relationship of these compiler sections to one another is shown in Figure 1.

The compiler, its storage, and its support system will require an allocation of at least 24,000 words of core storage and a minimum of five storage devices. If additional core storage is available at compile time, the compiler will use it.

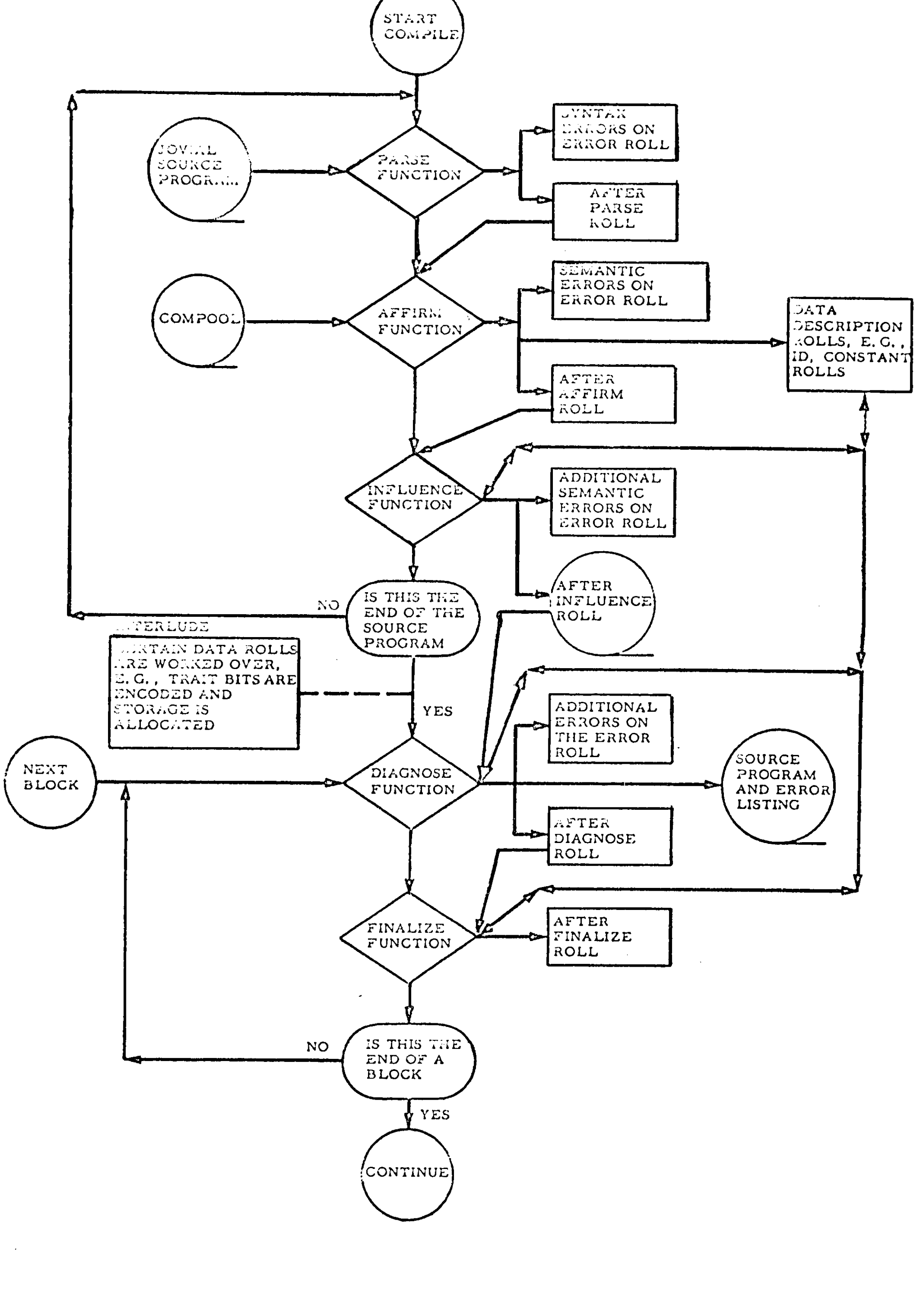

START COMPILE
JOVIAL SOURCE PROGRAM
PARSE FUNCTION
SYNTAX ERRORS ON ERROR ROLL
AFTER PARSE ROLL
COMPOOL
AFFIRM FUNCTION
SEMANTIC ERRORS ON ERROR ROLL
DATA DESCRIPTION ROLLS, E. G., ID, CONSTANT ROLLS
AFTER AFFIRM ROLL
INFLUENCE FUNCTION
ADDITIONAL SEMANTIC ERRORS ON ERROR ROLL
AFTER INFLUENCE ROLL
NO
IS THIS THE END OF THE SOURCE PROGRAM
INTERLUDE
CERTAIN DATA ROLLS ARE WORKED OVER, E. G., TRAIT BITS ARE ENCODED AND STORAGE IS ALLOCATED
YES
NEXT BLOCK
DIAGNOSE FUNCTION
ADDITIONAL ERRORS ON THE ERROR ROLL
SOURCE PROGRAM AND ERROR LISTING
AFTER DIAGNOSE ROLL
FINALIZE FUNCTION
AFTER FINALIZE ROLL
NO
IS THIS THE END OF A BLOCK
YES
CONTINUE

FIGURE 1 - OVERALL RELATIONSHIP OF COMPILER SECTIONS (Part 1)

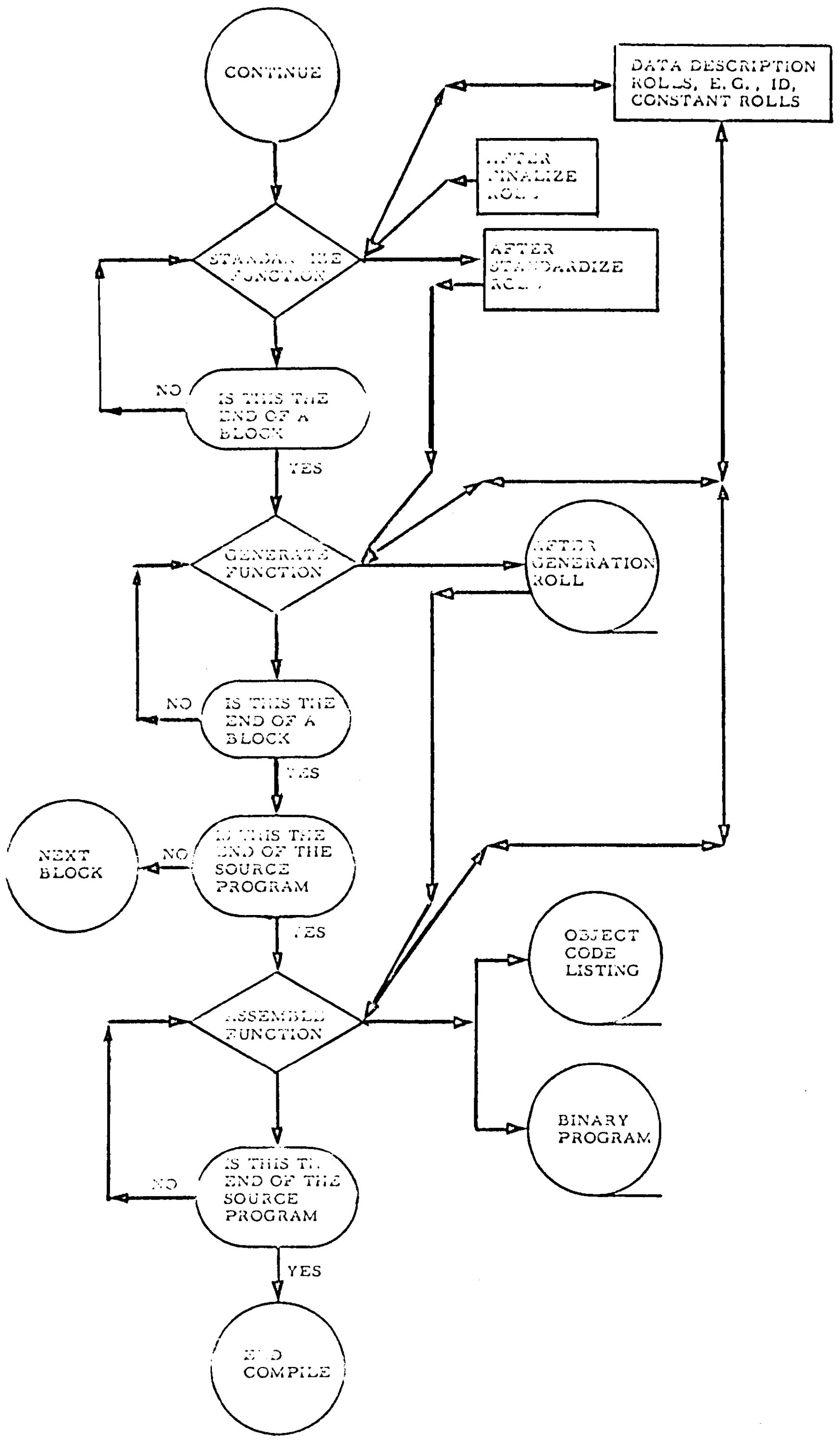


FIGURE 1 (continued) OVERALL RELATIONSHIP OF COMPILER SECTIONS (Part 2)

2. COMPILER INPUTS AND OUTPUTS

INPUT

Control Information

The JOVIAL compiler control parameters are input via the GE-625/635 General Comprehensive Operating Supervisor (GECOS) system control card used to initiate a JOVIAL compilation. This card has the following format:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
$							J	O	V	I	A	L			P	1	,	P	2	,	P	3	,	P	4	,	P	5											

where the possible parametric values, P1-P5 are:

DECK	Produce a binary output file for punching.
NOLISTIN	Do not produce a listing of source deck with error indications.
NOBUG	Do not include object code debug.
COMDK	Produce a compressed file from the input deck.
LISTOUT	Produce a listing of binary and symbolic output.

Any of these parameters may be present on or absent from the control card and any parameter may appear in any column, separated from the others by commas. If a parameter is omitted, the predetermined value assumed is the negative of that parameter.

An example of a JOVIAL control card is the following:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
$							J	O	V	I	A	L			D	E	C	K	,	N	O	L	I	S	T	I	N	,	L	I	S	T	O	U	T		

Source Language

Card Format

The JOVIAL source card is a free-field format in columns 1 to 72.
Columns 73 to 80 are reserved for deck identification and sequencing, and are ignored by the compiler.

Program Format

The first card in a JOVIAL program must be a START card. The last card must be a TERM$ card.

All data must be declared before being used in the JOVIAL program.

To adhere to GECOS specifications, a $ must never appear in column 1 of a source card.

Input Mode

Any symbolic storage medium may be used to contain a JOVIAL source deck, provided that medium is acceptable to GECOS.

A number of programs may be stacked for input for one compilation under one $ JOVIAL control card, in which case the parameters on the $ JOVIAL card apply to all programs in the stack and no $ EXECUTE card is allowed for the run.

Deck Identification

Immediately preceding the START card in the user's deck, there must be an identification card. The compiler uses this card to create a SYMDEF to be used by GELOAD. There are three types of source input to the compiler: a program (PROGRM), a procedure (PROCED), and data to generate a Compool (GENCOM). The format of the identification card for each of these is described below.

Program

Columns 1-6 contain PROGRM. Columns 8-13 may be blank or contain a name, left-justified to be used as the SYMDEF. If no name is present a SYMDEF of six periods is created.

Procedure

Columns 1-6 contain PROCED. No other identification is necessary. The name of the procedure is used as the SYMDEF.

Compool Generation

Columns 1-6 contain GENCOM. Columns 8-13 must contain, left-justified, the name of this Compool. A Compool is a communications pool established by the programmer and used by the compiler during subsequent program compilation. It is a data dictionary containing descriptions and absolute storage locations of items, tables, and other elements of the language that are used in common by the various programs. It should be noted that the compiler can generate a Compool, or extract data from an established Compool during a normal compilation. It cannot generate a Compool and compile a JOVIAL program during the same activity.

OUTPUT

Compiler Listings

Source Language Listing

The source listing consists of a one-statement-per-line replication of the JOVIAL input deck.

Each line of the listing has a number at the left margin. This line number is used later in object code debugging printouts and in the object code listing as a method of correlating object code and debugging output with source language statements.

In addition to the line number, the sequential card occurrence upon which this statement began will appear at the right margin of the listing. This number is for use with the ALTER feature in GEFRC.

The following example shows three input cards first as they appear in a JOVIAL deck and then as they appear on the source listing. They are the 369th, 370th, and 371st cards in the deck:

Card No.	Input Card Contents
369	FAAB. FOR C = 0, 1, NENT(FIFI) - 1$ BEGIN IF FINAL(C) EQ
370	HOLD AND FIRST(C) EQ DFIRST(C)$ BEGIN TEMPS(0) = C$
371	TEMPS(1) = TEMPS(1) + 1$ GOTO FBAA$ END END

Source Listing

```
602              FOR C = 0, 1, NENT(FIFI) - 1$                  369
603   BEGIN                                                     369
604              IF FINAL($C$) EQ HOLD AND FIRST($C$)
                 EQ DFIRST($C$)$                                369
605   BEGIN                                                     370
606              TEMPS($0$) = CS                                370
607              TEMPS($1$) = TEMPS($1$) + 1$                   370
608              GOTO FBAA$                                     371
609   END                                                       371
610   END                                                       371
```

Error messages that pertain to a specific statement follow the statement in the listing, and an error pointer appears directly below the erroneous part of the statement. Error messages are further described in Appendix C.

Object Code Listing

In addition to the source language listing, there is an object code listing that corresponds to the output of the compiler. The object code listing resembles an assembly program listing. Specifically, the following fields exist from left to right on the page.

- o Core location -- is relative to a base address of 0 since this is a relocatable binary deck.
- o Label field -- may or may not exist. If there is a label, it is the original JOVIAL label or a generated label of the form Gxxxxx where x is an integer.
- o Machine operation mnemonic.
- o Operand(s) -- conventions are the same as those that apply to labels.
- o Source statement line number.
- o Octal representation of binary.
- o Octal representation of relocatable bits.

Error Messages

Compiler error messages are of two general types: syntactical (construction) errors and semantic (usage) errors. The syntactical errors are detected during the first phase of compilation and are retained on the error roll for later printout. Semantic errors are found during subsequent phases of compilation. Appendix C contains a preliminary discussion of JOVIAL syntactical and semantic errors. As previously described under compiler listings, all errors which refer to an individual statement are printed below the erroneous part of the statement. Errors which cannot be detected at the exact time of occurrence -- that is, the BEGIN and END out of phase -- are noted when they become apparent.

The compiler suppresses the EXECUTE indicator to the operating system when a semantic error is encountered in the object code. Unless 200 or more errors of any kind are discovered, compilation continues to conclusion.

BINARY OUTPUT

On request, the compiler produces a relocatable binary file for the program just compiled. This file is in the column binary format and contains the information sufficient for loading by GELOAD as specified in the Relocatable Deck Description section of the GE-625/635 General Loader manual (CPB-1008).

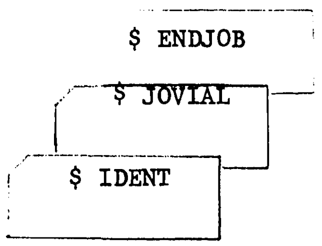
NUMERIC VALUES

Numeric values may be integer, fixed-point, or floating-point. Integer and fixed-point values may be carried in one or two words as necessary to conform to their declared size. Floating-point values are carried in one computer word.

3. OPERATING PROCEDURES

During checkout the GE-625/635 JOVIAL compiler is treated as an object program and is initiated by a $ EXECUTE card. After checkout the compiler operates as a system program under the control of GECOS and is initiated by a $ JOVIAL card.

When the compiler is operated as a system program, the following system control card configuration is a minimum requirement:



The inclusion of a $ EXECUTE card during a compilation run causes a bit to be set in the switch word. By examining this bit, the compiler determines whether or not a B*file is to be created. B* contains $ OBJECT followed by $ DKEND for Compool generation runs and for a noncompiling source deck.

Figure 2 illustrates the compiler's operating environment.

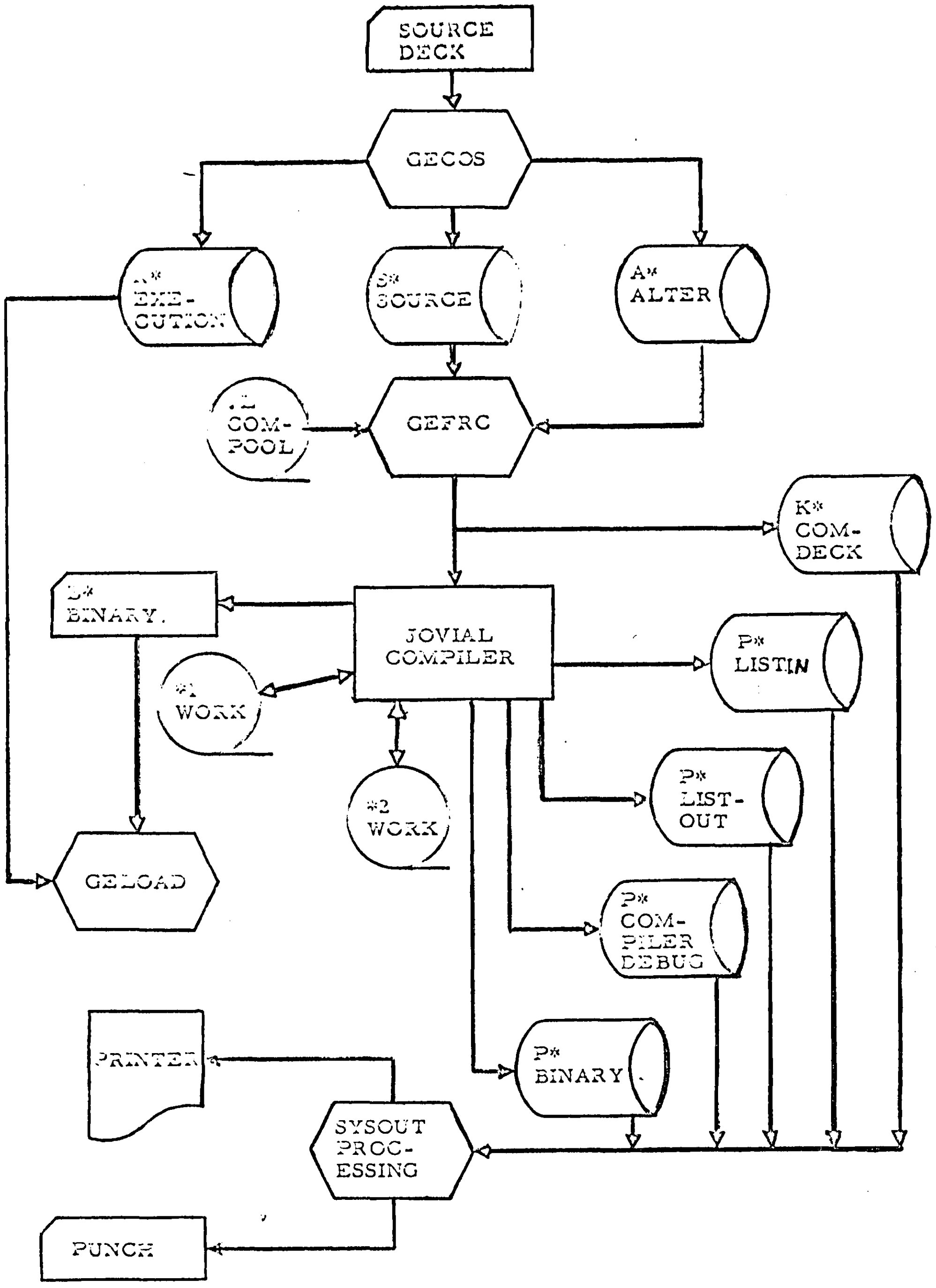


FIGURE 2 - OPERATING ENVIRONMENT FOR THE JOVIAL COMPILER

4. RESTRICTIONS

JOVIAL LANGUAGE RESTRICTIONS

The following are language variations in GE-625/635 JOVIAL:

- DUAL items are excluded
- STRING items are excluded
- Medium table packing is treated like dense table packing
- The J3X I/0 are implemented
- GE-625/635 GMAP assembly code (excluding all MACRO instructions) is the only legal type of direct code
- To conform with the conventions of GECOS, a $ must not appear in column 1 of an input card
- To conform with the conventions of GELOAD, procedure names must not be more than six characters

JOVIAL SOURCE PROGRAM SIZE RESTRICTIONS

The JOVIAL object program will always be compiled in a size that will operate within the same core allocation made to the compiler. Thus, if the compiler has allocated to it 24,000 words of core storage, it will compile a JOVIAL program whose expansion will occupy not more than 24,000 words of storage. For larger core allocations, all compiler rolls are dynamically allocated such that there is no wasted space and no need to penalize one program in favor of another.

A linear equation expresses the allocation of core in terms of compiler rolls:

$$C_1 I + C_2 D + \dots C_n L = \text{Available Roll Memory}$$

where $C_1, C_2, \dots, C_n$ are coefficients representing such things as roll group size and I, D, and L are the number of entries in a particular roll.

5. DIRECT CODE

The DIRECT statement in the JOVIAL language serves to define a set of operations which may be expressed in machine-oriented language. This language consists of GE-625/635 GMAP codes as defined below. Each line of machine code is considered a symbol, thus, comments, ASSIGN statements, and DIRECT or JOVIAL brackets must not appear on lines containing machine code instructions. Direct code may address locations designated by names defined in JOVIAL. A label of a direct code instruction, if constructed as a legal JOVIAL statement name becomes a defined statement name as though defined on a JOVIAL statement. A statement name prefixed to the DIRECT bracket is considered to designate the first executable instruction within the DIRECT JOVIAL brackets. Index registers used within DIRECT JOVIAL brackets should be saved and restored within those brackets.

INSTRUCTION REPERTOIRE

The GE-625/635 coding for use in DIRECT code are the instruction mnemonics listed in Appendix B of the GE 625/635 Programming Reference Manual (CPB-1004) with the following exceptions: LREG, SREG, STBA, and STBQ. In addition, the following pseudo codes are allowed: ZERO, SYMREF.

INSTRUCTION FORMAT

The machine code instructions within the DIRECT code must adhere to the following format:

Columns 1-6	label field

The label must be a legal GMAP label containing no special characters. If this label is to be referenced from a JOVIAL statement, the label must also adhere to legal JOVIAL label criteria.

Column 7	even-odd field

An E causes the instruction to be given an even address and an O causes the instruction to be given an odd address. A blank causes no significance to be given to addressing in terms of even or odd.

Columns 8-15	operation field

Instruction mnemonic begins in column 8.

Columns 16-72	location and comment field

This field has three subfields: subfield 1, subfield 2, and subfield 3.

- Subfield 1 may contain an operand.
- Subfield 2 may contain an operand or a modifier.
- Subfield 3 may contain comments.
- Subfield 1 and subfield 2 are separated by a comma and subfield 2 and subfield 3 are separated by a blank.

An operand may be a JOVIAL name or label or a legal DIRECT machine code label or a constant of the form d where d is a pure numeric decimal, or a constant of the form = 0 dd where dd is an octal number. Modifier is a legal GMAP modifier, i.e., DU. Any blank encountered in columns 16-72 terminates subfield 1 and subfield 2 and causes the remaining columns to be considered as comments.

Decimal increments or decrements are allowed in subfield 1 and subfield 2.

Either subfield may contain an asterisk which is translated as the address of the instruction of which this operand field is a part.

Special consideration is given to the instructions STCA and STCQ in the following manner. Subfield 2 must contain two octal characters whose bit configuration represent the character positions of the data referenced in subfield 1 which are stored.

6. OBJECT CODE INPUT/OUTPUT IMPLEMENTATION

INPUT/OUTPUT REQUESTS

The J3X JOVIAL I/O requests have the form:

IOn(FCN, FILENAME, XEC, DATANAME, PARAMETER) $

where	FCN	= code of operation to be performed
	FILENAME	= internally declared name of this file
	EXC	= name of a parameterless procedure to be executed upon completion of the I/O request
	DATANAME	= name of data storage area
	PARAMETER	= number of elements in data storage area
	IOn	= where n, a number of four digits or less, is the unique name of this particular request.

This information is processed by the compiler and used to generate a calling sequence to a subroutine called in at object program run time. This subroutine then causes the I/O request to be operated. The calling sequence to this subroutine has the following elements included:

FCN	= numeric code of operation to be performed
FILE number	= number associated with the FILENAME by a JOVIAL FILE declaration
DATA address	= address of DATANAME
PARAMETER	= number of words of data storage
FILE STATUS WORD	= address of a location in which to store status information about this I/O request

FILE CONTROL

The J3X JOVIAL I/O file control requests have the following form:

IN (FCN, FILENAME)
OUT (FCN, FILENAME)

where	FCN	= code of the operation, i.e., close or open, to be performed
	FILENAME	= internally declared name of this file

This information is processed by the compiler and used to generate a calling sequence to a subroutine called in at object program run time. This subroutine then does the bookkeeping needed to perform the file control and also causes any device movement necessary, i.e., rewind. The calling sequence to this subroutine includes the following elements:

FCN	= numeric code of operation to be performed
FILE number	= number associated with the FILENAME by a JOVIAL FILE declaration

INPUT/OUTPUT STATUS CHECKING

The JOVIAL FILE declaration includes a list of allowable status constants for the named file. The interfacing routine that causes the I/O request to be performed stores the status of the device into a status word upon completion of the I/O request. Upon encountering a status check request, a status constant corresponding to that particular request is compared to the defined bit configuration in the status word of the given file, thus providing the desired check. (The I/O status codes were not available at the time of printing of this manual.)

7. COMPILER INTERFACES

GECOS

The JOVIAL compiler operates as a system program under the control of GECOS. All control cards necessary to the operation of GECOS are used, including a $ JOVIAL card that signals the initiation of a JOVIAL compilation to GECOS. The compiler is in a binary relocatable format acceptable to GELOAD for loading into memory from a disc storage device known to GECOS.

At execution, the compiler obtains its control information from the switch word set up by GECOS from data on the $ JOVIAL control card.

All input and output requests from the compiler are processed by GEFRC using specifically formatted calling sequences rather than MACRO instructions.

The output of the compiler is in a relocatable binary format suitable for loading by GELOAD and contains specifically formatted calling sequences for I/O operations.

GECOS and JOVIAL requirements combine to utilize nine files during compilation. These are given below with their GECOS designators:

File	GECOS file code
Source input	S *
Alter deck	A *
Binary output to punch	P *
Binary output to operate	B *
Compool	L *
Comdeck	K *
Source listing and assembly listing	P *
Utility work file	* 1
Utility work file	* 2

COMPOOL

The GE-635 Compool is an optional input to the JOVIAL compiler. It provides a dictionary of system data and program variables. Upon encountering data not defined in the program, but included in the Compool available to the compiler, the definition of that data and any preset values assigned to it in the Compool are added to the compiler rolls as if it were internally declared. In actuality, there may be many Compools, but only one may be used with a given compilation.

The user specifies the option of a Compool by requesting the Compool file to be included in the configuration for compilation. The compiler then proceeds, based on the information in the file control block that tells whether the Compool file is present or absent.

The Compool format and the type of information that can be contained in the Compool are described in Appendix A.

The program that generates and assembles a Compool is a portion of the compiler with slight modifications. The compiler initiates a Compool generation upon encountering control information after normal loading.

MEMORY REQUIREMENTS

The JOVIAL compiler has the capability to adapt to various memory allocations of 24,000 words or more. Dynamic allocation of roll storage up to the limits specified at compile time makes this possible.

SUBROUTINE LIBRARY

The compiler has the capability to generate linkage between object code and the FORTRAN library. All calls to procedures that may be present in an object program or on a library file generate a fixed-format calling sequence compatible with the existing FORTRAN library.

8. COMPILER FUNCTIONS

There are nine major functions in the GE-625/635 JOVIAL compiler. The first compilation function, Parse, reduces the JOVIAL source statement input to a Polish notation form. Each succeeding function operates on this Polish representation of the program as described in the following paragraphs.

The Polish notation encoded statements are generally taken from the input roll of a function (the output roll of a previous function) and put on the work roll. There they are operated on by the function and then placed on an output roll. The input roll of each function is generally called the Under (function) roll and the output roll is generally called the After (function) roll. For example, the input to the Finalize function is on the After Diagnose (or Under Finalize) roll and the output of the Finalize function is on the After Finalize (or Under Standardize) roll.

All compiler functions process at least one JOVIAL statement before passing control to another function. Some functions process a whole block at one time. And one function --Assemble -- processes the program in its entirety.

PARSE FUNCTION

The Parse function constitutes the initial phase of the GE-625/635 JOVIAL compiler. It accepts JOVIAL source language input from a storage device via the input roll (Under Parse roll) and outputs the elements of each statement onto the After Parse roll. Its function is to analyze a statement; determine its type and syntactical correctness; remove any unneeded content, e.g., blanks, comments, etc; and reduce the statement to a Polish notation representation.

The Parse function processes one JOVIAL statement and then transfers control to the Affirm Function.

AFFIRM FUNCTION

The Affirm function is responsible for the further processing of a statement on the After Parse roll, now called the Under Affirm roll. The constant and variable operands that appear on the Under Affirm roll in literal form are removed and placed on data description rolls, such as the Constant and Identifier rolls. In addition, Affirm adds descriptive information concerning the occurrence and type of these operands to the data description rolls. On the After Affirm roll, the literal operands are replaced by pointers to the data description rolls.

The Affirm function processes one JOVIAL statement and then transfers control to the Influence function.

INFLUENCE FUNCTION

The Under Influence roll provides one statement to the Influence function. Influence will analyze the statement in order to provide contextual information concerning operand traits. When Influence has completed a statement, it will put it on the output roll where it can be spilled onto magnetic tape if required.

The Influence function processes one JOVIAL statement and returns control to Parse, unless it is the end of the source program. If it is the end of the program, the Interlude function is called.

INTERLUDE FUNCTION

The purpose of the Interlude function is to perform needed processing of the data description rolls. Specifically, the traits that were previously generated for Identifiers are encoded to simplify subsequent processing. Also, data storage is partially allocated for each data entry, e.g., Constant storage + 53.

The Interlude function transfers control to the Diagnose function.

DIAGNOSE FUNCTION

The source program listing is generated and output during the Diagnose function. The Diagnose function has two main sections: Type and Convert.

1. Type Section

   The Type section attaches type information to each operator and operand and detects any remaining source language errors. It uses trait information supplied by the previous functions to accomplish this task.

2. Convert Section

   The Convert section determines where arithmetic conversions are required in the object code and inserts the conversion drivers on the After Diagnose roll.

The Diagnose function processes one JOVIAL statement and transfers control to the Finalize function.

FINALIZE FUNCTION

The finalize function is a key function of the JOVIAL compiler. It prepares the JOVIAL program for object code generation. That is, it rearranges, optimizes, and computes additional information to facilitate code generation. The Finalize function consists of three main sections: Analysis, Unnest, and Collapse.

1. Analysis Section

   The Analysis section computes information concerning the type of an expression - for example, the fact that an expression is linear or nonlinear, constant or variable -- which will be important to the Generate function. This information is added to the existing Polish notation code.

2. Unnest Section

   The Unnest section takes expressions that must be computed and retained in temporary storage and locates them for proper code generation, that is, in front of other expressions of which they are a part.

3. Collapse Section

   The Collapse section recognizes duplicate expressions and replaces repeated occurrences with a reference to the initial expressions.

The Finalize function processes one JOVIAL statement and then transfers control back to the Diagnose function, unless it is the end of a block. If it is the end of a block, the transfer is to the Standardize function.

STANDARDIZE FUNCTION

The Standardize function processes an entire block at one time. It puts the results of the Collapse section of the Finalize function in standard Polish notation form and sets up the proper references to generate labels and temporary storage.

When the block is processed, the Standardize function transfers control to the Generate function.

GENERATE FUNCTION

The Generate function translates Polish notation code into GE-625/635 machine code. The Generate function is the first function in the compilation process in which the machine-independent Polish notation code is replaced by a machine-dependent code. The output roll from the Generate function can be spilled onto an intermediate file if required.

The Generate function processes an entire block of the program at one time. If this is the last block of the program, control is passed on to the Assemble function; otherwise, control is transferred back to the Diagnose function.

ASSEMBLE FUNCTION

The Assemble function is similar to the second pass of a normal GE-625/635 assembly program. It turns the entire program into a relocatable binary output. In addition, the Assemble function outputs the object code listing and the memory storage map.

APPENDIX A

COMPOOL DESCRIPTION

COMPOOL CONTENT

The JOVIAL Compool may contain any of the following types of information.

Program Descriptions

1. Program name
2. Program length

Table Descriptions

1. Table name
2. Variable or rigid length indicator
3. Number of entries
4. Parallel or serial structure indicator
5. Number of words per entry

Item Descriptions

1. Item name
2. Table name (if this item is in a table)
3. Item type
   - A-Fixed-point
   - B-Boolean
   - I- Integer
   - F-Floating-point
   - H-Hollerith
   - T-Standard Transmission
   - S-Status
4. Number of bits or bytes
5. Signed or unsigned (if applicable)
6. Number of fractional bits (A type only)
7. Word number within entry (for items within completely defined tables)
8. Initial bit position within the word (for items within completely defined tables)
9. Location of status constants in the status constant area (S type only)
10. Number of status constants associated with this item (S type only)

Status Constants

Status constants corresponding to all items or arrays declared as S type.

Array Descriptions

1. Array name
2. Array type (A, B, I, F, H, T, S as defined in 3 of Item Descriptions)
3. Number of dimensions
4. Size of each dimension
5. Signed or unsigned (if applicable)
6. Number of fractional bits (A type only)
7. Location of status constants in the status constant area (S type only)
8. Number of status constants associated with this array (S type only)

FORMAT

Compool Format

The Compool consists of a set of rolls like those that result from a normal compilation. The number and order of the rolls is a function of what is required by the compiler to express data and program declarations.

Compool File Format

The Compool file is one binary record containing the information generated by a Compool generation run.

OPERATING PROCEDURES

The operation of making a Compool file is initiated in the same manner as that for a JOVIAL compilation.

Input

The first card input to the JOVIAL compiler during a Compool generation run has GENCOM in columns 1 to 6. This signals the compiler to generate a Compool rather than to compile a program. Following the GENCOM card is the environment definition cards, whose content is described in the Compool Contents paragraph above. A TERMS$ card terminates the input deck.

The format of the Compool definition cards is identical to that of their counterpart, JOVIAL declarations.

Output

The normal output of a Compool generation run is (1) a binary file containing the Compool, and (2) a file for listing its contents.

Method

The Compool generation run is adapted from the first two phases of the JOVIAL compiler. Modifications are made to remove unneeded capabilities. The basic logic is that of the compiler.

Usage

The JOVIAL compiler is the ultimate user of the Compool information. Upon determining that a Compool file is present, the compiler takes data from that file, as it is referenced by the object program, and adds the data to the proper rolls.

The order of data reference for the compiler is local, global, and then Compool.

APPENDIX B

OBJECT CODE DEBUG

At the programmer's option, the JOVIAL compiler automatically includes debugging aids in the program being compiled. No control information is required from the programmer other than to specify the inclusion or exclusion of debugging (see Control Information under the Input paragraph of Chapter 2).

The compiler analyzes the source program as it is being compiled to determine the importance of variables, to recognize loops, and to understand the general flow of the program. Using this information it inserts code to provide for:

1. Dumps of key variables -- e.g., FOR variables, assignment variables, I/O variables -- as they change.

2. A program trace, which is a list of program transfers

3. A listing of procedure entrances and exits.

All debug information starts with the name of the program. The main program is signified by an asterisk. Each statement trace starts with the compiler-assigned statement number followed by a slash mark. In an assignment statement, this is followed by the variable name (subscripted if necessary) and the value assigned to it. For the FOR statements, the value of the induction variable is given for each iteration.

Tracing automatically terminates after each statement has been executed n times, where n is a system parameter. The format is (1) line number, (2) OFF, (3) the number of statements executed prior to termination, and (4) the number of statements executed during the trace-off mode. Automatic tracing resumes when statements which have not been previously executed n times are encountered.

IF, IFEITH, and SWITCH statements are followed by the values of their expressions. The final debug information supplied under post-mortem consists of the results of the last three executions of each statement. The format is (1) line number, (2) number of times the statement was executed, and (3) the last three values. Statements not executed three times show values in accordance with the number of executions.

APPENDIX C

COMPILER ERROR MESSAGES

The general form of the JOVIAL compiler error messages is described under the Output paragraph in Chapter 2. Each message is self-explanatory; that is, there are no error numbers that must be looked up in a supplementary error list.

Some types of errors that may be detected and corresponding error messages that are printed follow.

```
1.  ALPHA ($A, B)  =  TWODARRAY ($A$)$
               ↑                    ↑
       Syntax Error         Too Few Subscripts

2.  BETA  =  1.25A2 + 3H(XYZ)$
                       ↑
            Arithmetic Type Required

3.  IF ALPHA  =  BETA$
              ↑
        Syntax Error
```

Progress Is Our Most Important Product

GENERAL ELECTRIC

Computer Department • Phoenix, Arizona

LITHO IN U.S.A.