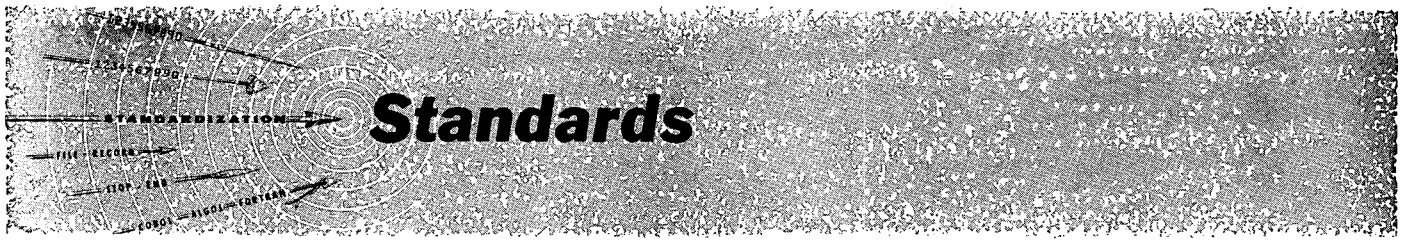

A Specification of JOVIAL*

CHRISTOPHER J. SHAW
System Development Corporation, Santa Monica, California

1. Introduction and Current Status

This report gives a complete specification of the latest "official" version[1] of JOVIAL, a general-purpose, procedure-oriented, and largely computer-independent programming language developed by System Development Corporation for large-scale military systems and as a corporate standard.

Work on JOVIAL, which is derived from ALGOL 58 [1] and from CLIP [2], began early in 1959. Since then, SDC has built JOVIAL compilers for several machines: the IBM 7090; the IBM AN/FSQ-31v (this compiler is also running on the closely related AN/FSQ-32); the IBM AN/FSQ-7; the Philco 2000; and the Control Data 1604 (this compiler has been adapted for use with the 1604A). The first two of these compilers accept an earlier and slightly different version of JOVIAL than is reported here. (And while all of the language reported here has been implemented in one compiler or another, none of them have implemented it entirely.)

In addition to the compilers mentioned above, a fast, one-pass compiler accepting a restricted subset of JOVIAL has been written and is running on the IBM 7090. It is being adapted for the AN/FSQ-32 as part of the time-sharing system being developed for that computer, and it is also being adapted for the Philco 2000. SDC is also writing a full-scale JOVIAL compiler for the IBM AN/FSQ-32 that should be in operation by the time this report appears.

JCVIAL has found application outside as well as inside SDC. Some thirty computer installations have received JOVIAL compilers, mainly through the users groups: SHARE, TUG and CO-OP. JOVIAL has been adopted by the Navy Command Systems Support Activity as the interim standard programming language for Navy strategic command systems.

2. Notation

In this report, the metalanguage used to describe JOVIAL syntax is the so-called Backus Normal Form,[2] with a few additions.

The elements of the metalanguage either denote or exhibit JOVIAL sign-strings. Except for the blank, JOVIAL signs thus stand for themselves, while terms made up of lower-case letters, possibly hyphenated, denote whole sets of JOVIAL sign-strings. The ⟨ and ⟩ brackets are used, not to enclose these terms, but to group strings of elements into, in effect, single elements.

A concatenation of these elements signifies a concatenation of JOVIAL signs. (Spaces have no meaning here and only improve readability.) The | symbol signifies selection between alternative strings of elements, (as limited by the ⟨ and ⟩ brackets.) The ::= symbol signifies syntactic equivalence.

A subscript appearing after any metalinguistic element is merely a "semantic" cue, with no formal syntactic effect.

The term *null* is introduced, with the following meaning:

null ::=
(The null or empty string of signs.)

To simplify the semantic explanation, alternative definitions of certain metalanguage elements are given at different places in the text. This has been noted, but the index at the end is perhaps the most convenient guide.

3. Alphabet and Vocabulary

JOVIAL's symbols are formed from an alphabet of 48 signs consisting of 26 letters, 10 numerals, and a dozen miscellaneous marks including the blank and the dollar

* Received August, 1963. This report owes much to the valuable help of M. H. Perstein and the other members of the Jovial Compiler Staff at System Development Corporation.

[1] The design of the language described in this report was completed in June 1961; no changes have been made since. Recently, however, it was decided again to consider language change proposals, and it can be expected that several minor improvements and extensions will have been adopted by the time this report is published.

[2] As used in 1963 [3].

sign. (This alphabet is the hardware alphabet as well as the reference alphabet.)

sign ::= letter | numeral | 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 ::= blank |**(|)|+|—|*|/|.|,|'|=|$**

Strings of JOVIAL signs form symbols, which are classed as delimiters, identifiers, and constants.

symbol ::= delimiter | identifier | constant

3.1 DELIMITERS. Delimiters are the verbs, the adjectives, and the punctuation of JOVIAL. They have fixed meanings, best described in later context.

delimiter ::= arithmetic-operator | relational-operator | logical-operator | sequential-operator | file-operator | functional-modifier | separator | bracket | declarator | descriptor
arithmetic-operator ::= **+|—|*|/|****
relational-operator ::= **EQ|GR|GQ|LQ|LS|NQ**
logical-operator ::= **AND|OR|NOT**
sequential-operator ::= **IF|GOTO|FOR|TEST|CLOSE|RETURN|STOP|IFEITH$_{er}$|ORIF**
file-operator ::= **OPEN|SHUT|INPUT|OUTPUT**
functional-modifier ::= **BIT|BYTE|MANT$_{issa}$|CHAR$_{acteristic}$|ODD|NENT|NWDSEN|ALL|ENTRY|POS$_{ition}$**
separator ::= **.|,|=|==|...|$**
bracket ::= **(|)|(/|/)|($|$)|"|"|BEGIN|END|DIRECT|JOVIAL|START|TERM**
declarator ::= **ITEM|MODE|ARRAY|TABLE|STRING|OVERLAY|DEFINE|SWITCH|PROC$_{edure}$|FILE**
descriptor ::= **F$_{loating}$|A$_{rithmetic}$|D$_{ual}$|S$_{igned}$|U$_{nsigned}$|R$_{ounded}$|H$_{ollerith}$|T$_{ransmission-code}$|S$_{tatus}$|B$_{oolean}$|V$_{ariable}$|R$_{igid}$|P$_{reset}$|L$_{ike}$|P$_{arallel}$|S$_{erial}$|D$_{ense}$|M$_{edium}$|N$_{o}$|B$_{inary}$**

3.2 IDENTIFIERS. Identifiers are either loop counters or names. Loop counters serve to identify the intrinsic set of signed, integer-valued variables used in controlling loops. (Loop counters are discussed further in Section 7 on Loops.) Names serve to identify the elements of a JOVIAL program's information environment: statements, switches, procedures, items, array-items, tables, string-items, and files. Except for context-defined statement names (and, in a sense, mode-defined item names), all JOVIAL names must be declared—either explicitly, in the program, or implicitly, in the Compool[3] or the procedure library.

A JOVIAL name is an arbitrary—though usually mnemonic—alphanumeric symbol, at least two characters long, which may be punctuated for readability by the ' mark.

identifier ::= loop-counter | name
loop-counter ::= letter
name ::= ⟨letter | name⟩ ⟨null | '⟩ ⟨letter | numeral⟩

Examples:

A
Z
STEP'01
BRANCH
U238
FLIGHT'POSITION

Names must obviously be distinguishable from delimiters and from each other. This is done by unique spelling—with the exception that names defined within a procedure are defined only for that procedure and do not conflict with names defined outside it. Furthermore, names defined within a program are defined only for that program, excluding procedures that define identically-spelled names. Names defined in a Compool are defined for an entire program system, excluding programs or procedures that define identically-spelled names. The names of data elements (items, arrays, tables, strings, and files) must be defined before they may be used.

3.3 CONSTANTS. JOVIAL programs manipulate four types of data: numeric values, consisting of the class of rational numbers and rational number pairs; literal values, consisting of strings of JOVIAL signs; status values, consisting of independent sets of arbitrarily named states (such as Good, Fair, Poor); and Boolean values, consisting of the two values, True and False.

constant ::= numeric-constant | literal-constant | status-constant | Boolean-constant
numeric-constant ::= integer-constant | floating-constant | fixed-constant | octal-constant | dual-constant

A JOVIAL constant, therefore, denotes a particular value as represented by a particular machine-language symbol. Numbers, integer constants, and floating- and fixed-point constants denote numeric values in the conventional, decimal sense, while octal constants have the obvious meaning of octal integers and dual constants denote pairs of numeric values. Literal constants denote Jovial sign-strings, represented in one of two possible 6-bit-per-sign encoding schemes, status constants are mnemonic names denoting qualities or categories rather than numeric values, and Boolean constants denote either True (by **1**) or False (by **0**).

number ::= numeral ⟨null | number⟩
signed ::= null | **+** | **—**
integer-constant ::= number | number **E**$_{exponent\text{-}base\text{-}10}$ number
floating-constant ::= ⟨number **.** | number **.** number | **.** number⟩ ⟨null | **E**$_{exponent\text{-}base\text{-}10}$ signed number⟩
fixed-constant ::= floating-constant **A** signed number $_{of\text{-}fraction\text{-}bits}$
octal-number ::= ⟨**0|1|2|3|4|5|6|7**⟩⟨null | octal-number⟩
octal-constant ::= **O**$_{ctal}$ **(** octal-number **)**
dual-constant ::= **D**$_{ual}$ **(** ⟨signed integer-constant **,** signed integer-constant | signed fixed-constant **,** signed fixed-constant | octal-constant **,** octal-constant⟩ **)**
sign-string ::= sign ⟨null | sign-string⟩
literal-constant ::= number$_{of\text{-}signs}$ **H**$_{ollerith\text{-}code}$ **(** sign-string **)** | number$_{of\text{-}signs}$ **T**$_{ransmission\text{-}code}$ **(** sign-string **)** | octal-constant
status-constant ::= **V**$_{alue}$ **(** ⟨letter | name⟩ **)**
Boolean-constant ::= **1**$_{true}$ | **0**$_{false}$

Examples:

018
123E4
.5
5.6789E—4A36
O(77760)
D(—32,+16)
27H(THIS IS A LITERAL CONSTANT.)
11T(SO IS THIS.)
V(EXCELLENT)
1

[3] A Compool, in effect, is a library of system environment declarations and storage allocation parameters.

In the integer-, floating-, and fixed-constants, the number following **E** is a decimal scaling factor expressed as an integral power of 10. In the fixed-constant, the number following **A** is a binary scaling factor, indicating precision, expressed as the number of fractional bits included in the machine-language symbol representing the value. (A negative precision-number implies truncation of least significant integral bits.)

Of the two coding schemes available for representing literal values, the more generally useful is Hollerith, the machine-dependent code by which literal values are input and output. Transmission-code, on the other hand, with its defined representation, does allow machine-independent symbol manipulation procedures to be written. The collating sequence for Transmission-code (wherein # denotes an unused numeric code) is the following: blank ##### **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) — + # = # # $ * (# # # #** , **# 0 1 2 3 4 5 6 7 8 9 ′ # / . # #**. Any JOVIAL sign may appear in a literal constant's sign-string, so the number of signs within the parentheses must equal the number preceding the **H** or **T**. Notice that an octal constant may denote a literal value (or, rather, its machine-language representation). This is useful, for example, in order to specify a code not associated with a JOVIAL sign.

The value of a status constant is defined only in context with a corresponding status variable, so its name may duplicate that of other status constants associated with different status variables. (Indeed, it may even duplicate a name used for another purpose.)

4. Comments

A comment allows a remark or clarifying text to be included among the symbols of a JOVIAL program. Comments are ignored by the compiler and so have no operational effect whatever on the program.

comment ::= " sign-string$_{\text{excluding-the-"-and-\$-symbols}}$"

Example:

"THIS IS A COMMENT."

Note that the omission of either " bracket is a major error, for subsequent commentary is interpreted by the compiler as program, and vice versa. To minimize the effect of such an error, the sentence-terminating **$** separator (though not the **($** and **$)** brackets) is excluded from commentary. Also excluded, of course, is the " bracket itself.

5. Clauses

Strings of JOVIAL symbols (delimiters, identifiers, and constants) form clauses: item descriptions, which describe values; variables, which designate values; and formulas, which specify values. In general, symbols may be separated by comments or by an arbitrary number of blanks.

separation ::= ⟨blank | comment⟩ ⟨null | separation⟩

However, no separation is needed when one or both of the signs so joined is a mark. To indicate symbol separation, then, the context-dependent term ∘ must be introduced, with the following meaning:

⟨numeral | letter⟩ ∘ ⟨numeral | letter⟩ ::= ⟨numeral | letter⟩ separation ⟨numeral | letter⟩

sign ∘ mark ::= sign ⟨null | separation⟩ mark

mark ∘ sign ::= mark ⟨null | separation⟩ sign

5.1 ITEM DESCRIPTIONS. In JOVIAL, the basic units of data are called items. All the necessary characteristics of an item's value, such as its type, and the format and coding of the machine-language symbol representing it, need be supplied only once—in an item description.

description ::= numeric-item-description | literal-item-description | status-item-description | boolean-item-description

numeric-item-description ::= floating-point-item-description | fixed-point-item-description | dual-item-description

floating-point-item-description ::= **F**$_{\text{loating}}$ ∘ ⟨null$_{\text{truncated}}$|**R**$_{\text{ounded}}$⟩ ∘ ⟨null | floating-constant ∘ ... ∘ floating-constant⟩

fixed-point-item-description ::= **A**$_{\text{rithmetic}}$ ∘ number$_{\text{of-bits}}$ ∘ ⟨**S**$_{\text{igned}}$ | **U**$_{\text{nsigned}}$⟩ ∘ ⟨null$_{\text{integral}}$ | signed number $_{\text{of-fractional-bits}}$⟩ ∘ ⟨null$_{\text{truncated}}$ | **R**$_{\text{ounded}}$⟩ ∘ ⟨null | integer-constant ∘ ... ∘ integer-constant | fixed-constant ∘ ... ∘ fixed-constant⟩

dual-item-description ::= **D**$_{\text{ual}}$ ∘ number$_{\text{of-bits-per-half}}$ ∘ ⟨**S**$_{\text{igned}}$ | **U**$_{\text{nsigned}}$⟩ ∘ ⟨null$_{\text{integral}}$ | signed number$_{\text{of-fractional-bits-per-half}}$⟩ ∘ ⟨null$_{\text{truncated}}$ | **R**$_{\text{ounded}}$⟩ ∘ ⟨null | dual-constant ∘ ... ∘ dual-constant⟩

literal-item-description ::= ⟨**H**$_{\text{ollerith-code}}$ | **T**$_{\text{ransmission-code}}$⟩ ∘ number$_{\text{of-signs}}$

status-list ::= status-constant ∘ ⟨null | status-list⟩

status-item-description ::= **S**$_{\text{tatus}}$ ∘ ⟨null | number$_{\text{of-bits}}$⟩ ∘ status-list

boolean-item-description ::= **B**$_{\text{oolean}}$

Examples:

F R .5E—7 ... .5E+3
A 36 S 15
D 16 U R D(0,0) ... D(25E3, 25E3)
H 120
S V(BAD) V(POOR) V(FAIR) V(GOOD) V(FINE)
B

In the three numeric item descriptions, the **R** for *R*ounded descriptor declares that any value assigned to the item be rounded rather than truncated, as would be the case were the **R** omitted. The pair of numeric constants, separated by the **...** separator, declare the estimated minimum and maximum absolute values of the item, in that order. (This optional, estimated range is useful mainly for purposes of program documentation, though it may be used by a compiler to optimize the machine-language program's manipulation of the item.)

In both fixed-point and dual item descriptions, number of bits includes both integral and fractional bits, and a sign-bit, if any. (The presence of a sign-bit is declared by the **S** for *S*igned descriptor; the absence, by the **U** for *U*nsigned descriptor.) If number of fractional bits is omitted, an exact integer (or dual integer) is declared; and if number of fractional bits is negative, the low-order integer bits are declared not significant and therefore need not be represented.

In the status item description, the list of status constants declares the possible values of the item. These values are encoded, in order, by the series of numbers 0, 1, 2, 3, etc. When number of bits is omitted, the size of the machine-language symbol needed to represent the item's value is

derived from the number of status constants. (When a number of bits k is given, the number of status constants may not exceed 2^k.)

5.2 VARIABLES. A variable designates a value which may be altered during the course of program execution. Since items are the basic units of data in JOVIAL, they are the chief variables. If an item name designates more than one value, as in an array, table, or string, then an index list (of numeric formulas) must be appended (as a subscript, enclosed in the **($** and **$)** brackets) to distinguish a particular value.

variable ::= numeric-variable | literal-variable | status-variable | Boolean-variable
index ::= numeric-formula
index-list ::= index ∘ ⟨null | , ∘ index-list⟩
subscript ::= **($** ∘ index-list ∘ **$)**
numeric-variable ::= name$_{\text{of-numeric-item}}$ ∘ ⟨null | subscript⟩
literal-variable ::= name$_{\text{of-literal-item}}$ ∘ ⟨null | subscript⟩
status-variable ::= name$_{\text{of-status-item}}$ ∘ ⟨null | subscript⟩
boolean-variable ::= name$_{\text{of-Boolean-item}}$ ∘ ⟨null | subscript⟩

JOVIAL has other variables besides items. These are discussed in the sections on loops (7), functional modifiers (19), and input-output (13).

5.3 FORMULAS. A formula specifies a value and is, in effect, a computing rule for obtaining that value. A formula may contain variables and so the value it specifies, in general, is dynamically dependent on these variables, as will be described.

formula ::= numeric-formula | literal-formula | status-formula | boolean-formula

5.3.1 *Functions.* A function specifies the value computed by a procedure utilizing the function's actual input parameters, if any. (These are values, as specified by formula, and arrays or tables, as denoted by name.) Functions are numeric, literal, status, or Boolean—according to the type of data value they specify. (The rules governing actual parameters and function types are covered in the section on procedures.)

function ::= name$_{\text{of-procedure}}$ ∘ **(** ∘ ⟨null | actual-input-parameter-list⟩ ∘ **)**
Examples:
ARCSIN (GAMMA*2.72, 1.0E—4)
RANDOM ( )
SYMMETRIC(MATRIX'A)

Because a function value is computed by a procedure, specifying a function value may have other "side-effects" on program execution. However, the value specified by a formula is undefined when that formula contains both a variable, and a function that affects its value.

5.3.2 *Numeric Formulas.* A numeric formula specifies a numeric value computed from the values expressed by its individual operands—numeric constants, variables, and functions. The arithmetic operators **+**, **—**, *, /, and ** have the conventional algebraic meanings of addition, subtraction or negation, multiplication, division, and exponentiation. As in algebra, division by zero is undefined. Fractional or mixed exponents are possible, but since JOVIAL deals only with rational numbers, any exponentiation that would specify a complex root, such as **(—2)**.5**, is also undefined. The parentheses **(** and **)** perform their usual grouping function, and the absolute value brackets **(/** and **/)** specify the magnitude of the value of the numeric formula they enclose. With these brackets, formulas of any complexity may be constructed.

numeric-formula ::= numeric-constant | numeric-variable | function$_{\text{of-numeric-type}}$ | ⟨**+** | **—**⟩ ∘ numeric-formula | **(** ∘ numeric-formula ∘ **)** | **(/** ∘ numeric-formula ∘ **/)** | numeric-formula ∘ arithmetic-operator ∘ numeric-formula
Examples:
AA(0)2/(AA(1)—AA(2)/(XX+1/XX))**
(—273.*ALPHA(0)+(/BETA($T2$)/)—
LOG(BETA($T2$)))/1.889E—6

The sequence of arithmetic operations in a numeric formula is determined primarily by the way the formula is bracketed and secondarily by the conventional operator precedence scheme: first, negations are performed; second, exponentiations; third, multiplications and divisions; fourth, additions and subtractions; finally, within these categories, operations are performed from left to right in order of listing.

In JOVIAL, numeric values have three modes of representation: floating-point, fixed-point, and dual fixed-point. Any arithmetic operation may be performed in any one of these modes, upon operands of like mode. (In the dual mode, operations are done in parallel, with the left component of one operand combined with the left component of the other to yield the left component of the result, and similarly for right components.) However, a numeric formula may combine operands having different modes of representation, so the following automatic conversions between modes are implied: fixed to floating, floating to fixed, fixed to dual.[4]

The precision of the result of a fixed-point (or dual fixed-point) computation is compiler-dependent and cannot be exactly defined, but some useful limits can be established. Where the maximum possible significance of a result exceeds the maximum allowed numeric-operand size, the result is truncated to this limit in the following manner: first, the *least* significant fraction bits of the result are truncated; and second, if necessary, the *most* significant integer bits. (In determining the maximum possible significance of a fixed-point result, exact integers are regarded as arbitrarily precise.)

5.3.3 *Literal and Status Formulas.* Both literal and status formulas specify a value as expressed by a single operand—a constant, a variable, or a function—of the appropriate type.

literal-formula ::= literal-constant | literal-variable | function$_{\text{of-literal-type}}$
status-formula ::= status-constant | status-variable | function$_{\text{of-status-type}}$

[4] The single-valued fixed-point operand is "twinned" or duplicated.

Examples:

```
6T(ABACUS)
O(060706103230)
SIGNAL
V(CLOUDY)
WEATHER($AIRBASE$)
TYPE (SIGNAL)
```

5.3.4 *Boolean Formulas.* A Boolean formula specifies a Boolean value, either True or False, computed from the values expressed by its individual operands—Boolean constants, variables, and functions; and relational formulas.

numeric-relation-list ::= relational-operator ∘ numeric-formula ∘ ⟨null | numeric-relation-list⟩

literal-relation-list ::= relational-operator ∘ literal-formula ∘ ⟨null | literal-relation-list⟩

relational-formula ::= numeric-formula ∘ numeric-relation-list | literal-formula ∘ literal-relation-list | ⟨status-variable | $\text{name}_{\text{of-file}}$⟩ ∘ relational-operator ∘ status-formula

boolean-formula ::= boolean-constant | boolean-variable | $\text{function}_{\text{of-boolean-type}}$ | relational-formula | (∘ boolean-formula ∘) | **NOT** ∘ boolean-formula | boolean-formula ∘ ⟨**AND** | **OR**⟩ ∘ boolean-formula

Examples:

```
0
INDICATOR
LEGAL (SIGNAL)
-13 LS ALPHA LQ +100 LS BETA($T2$) LQ +198
1T(A) LQ SIGNAL LQ 1T(Z)
WEATHER($AIRBASE$) EQ V(FAIR)
INDICATOR AND NOT (WEATHER($AIRBASE$) EQ V(FAIR)
  OR LEGAL(SIGNAL))
```

A relational operator compares the pair of values specified by the formulas on either side to determine whether the indicated relation holds between them. A relational formula thus specifies True only when *all* its relations hold. The relational operators indicate primarily numeric relations: **EQ**, is EQual to; **NQ**, is uNeQual to; **GR**, is GReater than; **LQ**, is Less than or eQual to; **LS**, is LesS than; and **GQ**, is Greater than or eQual to. They may be used, however, to compare both literal and status values on the basis of their numeric encoding. Shorter literal values are prefixed by blanks before comparison. (The use of a file name in a relational formula is treated in Section 13.2, Testing and Positioning Files.)

Logical operations can be performed on Boolean values in much the same way that arithmetic operations are performed on numeric values. The logical operator **NOT** reverses the value specified by the subsequent Boolean formula, while **AND** yields True only if the Boolean formulas on either side both specify True, and **OR** yields False only if the Boolean formulas on either side both specify False. Unless parentheses indicate otherwise, the precedence of the logical operations is: **NOT**'s first, **AND**'s second, and **OR**'s last; from left to right.

6. Basic Declarations and Statements

Clauses are combined with certain delimiters to form declarations and statements, which are the sentences of JOVIAL. Statements assert actions that the program is to perform (normally in the sequence in which they are listed) and declarations describe the information environment in which the actions are to occur.

declaration ::= item-declaration | mode-declaration | array-declaration | table-declaration | initial-value-declaration | overlay-declaration | define-declaration | procedure-declaration | switch-declaration | file-declaration

6.1 ITEM DECLARATIONS. In data processing, the natural unit of information is the value. In JOVIAL, values other than those denoted by constants, or used only as intermediate results, or for controlling loops, must be formally declared as items—simple items, array items, table items, or string items—before they may be referenced. When not part of a table declaration, an item declaration defines a simple item, with a single value.

item-declaration ::= **ITEM** ∘ $\text{name}_{\text{of-item}}$ ∘ description ∘ **$**

Examples:

```
ITEM P66 F $
ITEM TALLY A 15 U R 0 ... 2E4 $
ITEM FLIGHT'POSITION D 16 S 5 $
ITEM IDENT H 12 $
ITEM HEADING S 6 V(N) V(NE) V(E) V(SE) V(S) V(SW)
  V(W) V(NW) $
ITEM SPARE B $
```

6.2 MODE DECLARATIONS. A mode declaration starts a new normal mode of item description for the implicit declaration of *all* subsequently referenced and otherwise undefined simple items.[5]

mode-declaration ::= **MODE** ∘ description ∘ **$**

Examples:

```
MODE F R $
MODE A 15 U 3 $
MODE B $
```

After a mode declaration, the initial occurrence of any name, in any context where a simple item name is expected, serves at that point to declare an item with that name, described according to the mode. A mode declaration remains effective until superseded by another mode declaration.

6.3 ARRAY DECLARATIONS. An array declaration describes the structure of a collection of similar item values, and also provides a means of identifying this collection with a single item name. Rectangular arrays of any number of dimensions may thus be declared.

dimension-list ::= $\text{number}_{\text{of-items-per-dimension}}$ ∘ ⟨null | dimension-list⟩

array-declaration ::= **ARRAY** ∘ $\text{name}_{\text{of-item}}$ ∘ dimension-list ∘ description ∘ **$**

Examples:

```
ARRAY ALPHA 24 3 5 F R $
ARRAY CARD'IMAGE 80 H 1 $
ARRAY TIC'TAC'TOE 3 3 S V(EMPTY) V(NOUGHT)
  V(CROSS) $
ARRAY CHARACTER'MATRIX 7 5 B $
```

In designating an individual value from an n-dimensional array, the array item name must be subscripted by an n-component index list of numeric formulas; and where the size of a dimension is k items, the integral value

[5] The *initial*, normal mode of item description is compiler-dependent.

(truncated, if necessary) of the corresponding component of the index list can only range from 0 thru $k-1$.

6.4 Simple, Complex, and Compound Statements. It is convenient to recognize three types of statements in Jovial: simple statements, which express "primitive" data processing actions; complex statements, which incorporate simple or compound statements within them; and compound statements, which group together whole strings of statements—simple, complex, or compound.

statement ::= simple-statement | complex-statement | compound-statement
statement-list ::= ⟨statement | declaration⟩ ∘ ⟨null | statement-list⟩
simple-statement ::= assignment-statement | exchange-statement | go-to-statement | test-statement | stop-statement | return-statement | procedure-statement | input-statement | output-statement
complex-statement ::= conditional-statement | loop-statement | direct-code-statement | alternative-statement | closed-statement
compound-statement ::= **BEGIN** ∘ statement-list ∘ **END**

Notice that a compound statement may contain declarations. (However, the scope of these declarations is generally not limited by the **BEGIN** and **END** brackets that serve as statement parentheses.) To function as a statement, though, a compound statement must contain at least one statement that is not a closed statement. (The various simple and complex statements are treated in subsequent sections.)

6.5 Named Statements. A statement must often be named to permit it to be executed out of its normal, listed sequence. Any Jovial statement—simple, complex, compound, or already named—may be named. A name is needed, however, only when the statement is to be executed out of sequence.

simple-statement ::= name$_{\text{of-statement}}$ ∘ **.** ∘ simple-statement
complex-statement ::= name$_{\text{of-statement}}$ ∘ **.** ∘ complex-statement
compound-statement ::= name$_{\text{of-statement}}$ ∘ **.** ∘ compound-statement

6.6 Assignment Statements. An assignment statement assigns the value specified by a formula to be the value thereafter designated by a variable. The formula must, therefore, specify a value of the type—numeric, literal, status, or Boolean—designated by the variable.

assignment-statement ::= variable ∘ **=** ∘ formula ∘ **$**
Examples:
```
ALPHA = ALPHA + 27 $
SIGNAL = 1T(M) $
WEATHER($AIRBASE$) = V(CLOUDY) $
INDICATOR = -13 LS ALPHA LQ +100 LS BETA LQ +198
  OR LEGAL (SIGNAL) $
```

During assignment, where necessary, numeric values are automatically converted to the representation, and are rounded or truncated to the precision, of the variable to which they are being assigned. However, the following are undefined: assigning a negative value to an unsigned variable; assigning a dual value to a fixed- or floating-point variable; assigning a value to a variable with fewer significant bits than the value.

Literal values are right-justified during assignment and, where necessary, they are prefixed by blanks. However, assigning a literal value to a variable with fewer signs than the value is undefined.

Status value assignments operate as if on unsigned integers, but assigning a status value to a variable with a different encoding is undefined.

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

exchange-statement ::= variable ∘ **==** ∘ variable ∘ **$**
Examples:
```
SIGNAL == CARD'IMAGE($275$) $
WEATHER($AIRBASE$) == WEATHER($AIRBASE+1$) $
```

6.8 Conditional Statements. A conditional statement is a complex statement whose two major constituents are an if-clause containing a Boolean formula and a simple or compound statement. The execution of this statement is conditional. If the Boolean formula specifies True, the statement is executed. If it specifies False, the statement is skipped, and operation continues with the next listed statement.

if-clause ::= **IF** ∘ boolean-formula ∘ **$**
conditional-statement ::= if-clause ∘ ⟨simple-statement | compound-statement⟩

Example, contained in a (compound) Jovial statement computing gross pay for an hourly employee:

```
BEGIN COMPUTE'GROSS'PAY.
GROSS'PAY($EMPLOYEE$) = HOURS'WORKED($EMPLOY-
  EE$) * HOURLY'PAY($EMPLOYEE$) $
IF HOURS'WORKED($EMPLOYEE$) GR 40 $ GROSS'PAY
  ($EMPLOYEE$) = GROSS'PAY($EMPLOYEE$) + (HOURS'
  WORKED($EMPLOYEE$) - 40) * HOURLY'PAY($EM-
  PLOYEE$) / 2 $
END "COMPUTE'GROSS'PAY"
```

6.9 Go-To Statements. A go-to statement breaks the normal, listed sequence of statement executions by transferring control to the statement bearing the given name, or specified by the switch call. (Switches and switch calls are treated in Section 12.)

go-to-statement ::= **GOTO** ∘ ⟨name$_{\text{of-next-statement}}$ | switch-call⟩ ∘ **$**
Examples:
```
GOTO COMPUTE'GROSS'PAY $
GOTO STEP ($3$) $
```

7. Loops

Counting and count-controlled loops are among the most common operations in programming. For this reason, Jovial includes an intrinsic set of loop counters—signed, integer-valued variables, each designated by a single letter.

numeric-variable ::= loop-counter

A loop counter is activated and assigned an initial value only by the execution of a for-clause in a loop statement.

7.1 Loop Statements. A loop statement is a complex statement consisting of a list of for-clauses which establish the loop counters, and a simple or compound statement, which forms the repeatedly-executed body of the loop.

loop-indices ::= $index_{initial\text{-}value}$ | $index_{initial\text{-}value}$ ∘ , ∘ $index_{increment}$ | $index_{initial\text{-}value}$ ∘ , ∘ $index_{increment}$ ∘ , ∘ $index_{limit}$
for-clause ::= **FOR** ∘ loop-counter ∘ **=** ∘ loop-indices ∘ **$**
loop-statement ::= for-clause ∘ ⟨simple-statement | compound-statement | **BEGIN** ∘ statement-list ∘ if-clause ∘ **END** | loop-statement⟩

Examples. The first example sets the first 500 NUMBERs to zero; the second computes $\alpha_i = 2^{i+1}-1$; the third transposes the 25 by 25 matrix, **NODE.**

```
FOR A = 0,1,499 $ NUMBER($A$) = 0. $
FOR I = 0,1,99 $ FOR S = 1,S+1 $ ALPHA($I$) = S $
BEGIN TRANSPOSE. FOR I = 0,1,23 $
  BEGIN FOR J = I+1,1,24 $
    NODE($I,J$) == NODE($J,I$) $
END END
```

A loop statement activates one or more loop counters, assigns them each an initial value, and causes its constituent simple or compound statement to be executed one or more times. If the for-clauses in the loop statement all contain only initial-value indexes, this statement is executed just once. However, if any of the for-clauses contain an increment index, the statement is repeatedly executed, and after each repetition, the values of the appropriate loop counters are modified by the addition of the current values of the corresponding increment indices. (This occurs in reverse of the order in which the loop counters are activated.) Finally, one of the for-clauses may contain a limit index as well. (Only one such for-clause is allowed per loop statement,[6] and it must precede any other for-clauses containing increment indices.) If so, then the just-incremented value of the corresponding loop counter is tested, and the loop is terminated when that value exceeds the current value of the limit index—in the positive direction if the increment was positive or zero, and in the negative direction if the increment was negative.

The loop statement containing an if-clause just before the final **END** bracket conditionally performs the incrementing and limit testing routine. If the Boolean formula of the if-clause specifies the value True, the routine is performed. Otherwise, it is skipped and the loop is terminated.

The range of activity of a loop counter includes the for-clause whose execution activates it, all subsequent for-clauses, and the statement that forms the loop's repeatedly-executed body. The use of an active loop counter as a numeric variable is not restricted.

In general, only an inactive loop counter may be activated by a for-clause. An active loop counter may be reactivated, though, by the re-execution of its corresponding for-clause as the result of a transfer of control from within the loop. Individual for-clauses in a loop statement can be given statement names for this purpose. Ordinarily, however, any transfer of control into a loop statement from outside will produce undefined results, due to inactive (and therefore undefined) loop counters.[7]

7.2 Test Statements. The test statement, which is only defined within a loop statement, expresses a transfer of control to the implicit (and thus unnamed) loop-counter modification routine at the loop's end.

test-statement ::= **TEST** ∘ **$** | **TEST** ∘ loop-counter ∘ **$**
Examples:

```
TEST $
TEST Y $
```

A test statement with no loop counter indicated transfers control to the *first* loop counter modification of the innermost applicable loop statement, and thus effects the modification of all loop counters activated by that loop statement. On the other hand, if an active loop counter is indicated, control goes to the routine modifying that particular loop counter, so the modification of subsequently-activated loop counters is omitted. In either case, if the loop statement includes a limit formula, the loop-termination test is made.

8. Tables

A table is a matrix of item values. The rows of a table are called entries, and an entry consists of the values of a related set of different items. Typically, entry **K** (**ITEM'1(K), ITEM'2(K), ..., ITEM'N(K)**) would consist of values measuring the **N** pertinent attributes of "object" **K**. Such an entry would be associated with other, similar entries in a table, or list of entries.

All entries of a table usually have the same composition and structure in the sense that each consists of a similarly named and ordered set of index-related items, declared as part of the table declaration. Variable entry structures can be achieved, however, by using the technique of overlaying items and by the inclusion of string items in the entry.

table-declaration ::= ordinary-table-declaration | like-table-declaration | specified-entry-structure-table-declaration

8.1 Ordinary Table Declarations. A table is declared by a table declaration that includes an entry list of item declarations, enclosed in **BEGIN** and **END** brackets. These declare the items comprising an entry. Any of these items may be assigned initial values by including a list of constants after the item declaration, and they may be allocated storage common to other items in the entry by including overlay declarations. (These two topics will be covered in the sections on initial value declarations and overlay declarations.)

An individual value of a table item is designated by item name and entry index, and where a table has *k*

[6] This means that a loop statement can produce just a single loop. Loops within loops must be constructed by embedding the inner loop in a compound statement iterated by the outer loop.

[7] Transfer of control into a loop statement will bypass the execution of at least one for-clause, so corresponding loop counters will be inactive and their values undefined over the rest of the loop statement.

entries the numeric formula specifying entry index may only range in value from 0 through $k-1$.

table-type ::= **V**$_{\text{ariable-length}}$ | **R**$_{\text{igid-length}}$
entry-type ::= **S**$_{\text{erial-entry-structure}}$ | **P**$_{\text{arallel-entry-structure}}$
packing-mode ::= **N**$_{\text{o-item-packing}}$ | **M**$_{\text{edium-item-packing}}$ | **D**$_{\text{ense-item-packing}}$
entry-list ::= item-declaration ∘ ⟨null | **BEGIN** ∘ 1-dimensional-constant-list ∘ **END**⟩ | entry-list ∘ ⟨entry-list | overlay-declaration⟩
ordinary-table-declaration ::= **TABLE** ∘ ⟨null | name$_{\text{of-table}}$⟩ ∘ table-type ∘ number$_{\text{of-entries}}$ ∘ ⟨null | entry-type⟩ ∘ ⟨null | packing-mode⟩ ∘ **$** ∘ **BEGIN** ∘ entry-list ∘ **END**

Example:

```
TABLE  AIRBASE'WEATHER R 85 S D $
                       BEGIN
ITEM        AIRBASE'CODE H 3 "LETTERS" $
ITEM        REPORT'HOUR A 5 U 0...23 "HOURS" $
ITEM      REPORT'MINUTE A 6 U 0...59 "MINUTES" $
ITEM     WEATHER'CHANGE B "TRUE IF CHANGED
                          FROM LAST REPORT" $
ITEM  CURRENT'SUMMARY S V(OPEN) V(INSTRUMENT)
                          V(CLOSED) $
ITEM FORECAST'SUMMARY S V(OPEN) V(INSTRUMENT)
                          V(CLOSED) $
ITEM           CEILING A 9 U 0...511 "HUNDRED
                          FEET. MAXIMUM OF 511
                          MEANS UNLIMITED" $
ITEM        VISIBILITY A 5 U 1 0.A1...15.5A1 "NAUTI-
                          CAL MILES. MAXIMUM
                          OF 15.5 MEANS UNLIM-
                          ITED" $
ITEM    VISIBILITY'BLOCK S V(NONE) V(FOG) V(DUST)
                          V(SMOKE) V(HAZE) $
ITEM      BLOCK'AMOUNT S V(NONE) V(LIGHT)
                          V(MODERATE) V(HEAVY)
                          $
ITEM     PRECIPITATION S V(NONE) V(RAIN)
                          V(SNOW) V(SLEET)
                          V(HAIL) $
ITEM      PRECIP'AMOUNT S V(NONE) V(LIGHT)
                          V(MODERATE) V(HEAVY)
                          $
ITEM RUNWAY'CONDITION S V(OK) V(WET) V(ICY)
                          V(SNOW) V(BLOCKED) $
                       END
```

A table name may be omitted from the declaration when only individual table items are referred to in the program, and never the entire table.

The **V** for *V*ariable or **R** for *R*igid table-length descriptor indicates table type, and determines whether the number of entries can vary during program execution.

Number of entries indicates the table's maximum length for a variable-length table, and its fixed length for a rigid-length table.

The **S** for *S*erial or **P** for *P*arallel entry-structure descriptor indicates entry type and determines one of two possible storage configurations for the table. (If entry type is omitted, the compiler-dependent, "normal" entry type is assumed.) A k-word, serial-type entry is allocated a block of k serial, or consecutive, storage registers. A k-word, parallel-type entry, on the other hand, is allocated parallel, or similarly-located, storage registers in k separate blocks.

The **N** for *N*o, **M** for *M*edium, or **D** for *D*ense item-packing descriptor indicates a mode of storage allocation for the items in an entry. (If packing mode is omitted, the compiler-dependent, "normal" packing mode is assumed.) No packing means that items are allocated storage in full register units, so that each item in the entry will occupy one or more consecutive registers; medium packing means that storage is allocated in sub-register[8] units, with each item stored in one or more consecutive sub-registers; dense packing, finally, means that storage is allocated primarily in bit-position units, so that each item occupies one or more consecutive bit positions.

8.2 Like Table Declarations. In some cases, a program's environment must contain two or more instances of tables with the same entry structure. Such tables may be declared and named, using a previously-defined table as a pattern, with a like table declaration, by adding a distinguishing letter or numeral to the pattern table's name.

like-table-declaration ::= **TABLE** ∘ name$_{\text{of-pattern-table}}$ ⟨letter | numeral⟩ ∘ ⟨null | table-type ∘ number$_{\text{of-entries}}$⟩ ∘ ⟨null | entry-type⟩ ∘ ⟨null | packing-mode⟩ ∘ **L**$_{\text{ike}}$ ∘ **$**

Examples:

```
TABLE AIRBASE'WEATHER0 L $
TABLE AIRBASE'WEATHERX R 1 N L $
```

The composition and structure of the like table's entries are taken as being generated by the declarations describing the pattern table's entries, with the exception that all item names are suffixed with the distinguishing letter or numeral.[9] The like table may have its own descriptions of type and length, entry structure, and item packing, however, or by omission, it may retain those of the pattern table.

8.3 Specified Entry-structure Table Declarations. It is often necessary to declare a table with a specific and predetermined (and even variable) entry structure. The specified entry-structure table declaration provides complete control over the structure of table entries by means of the structured item declaration and the string item declaration.

structured-item-declaration ::= **ITEM** ∘ name$_{\text{of-item}}$ ∘ description ∘ number$_{\text{word-index}}$ ∘ number$_{\text{bit-index}}$ ∘ ⟨null | packing-mode⟩ ∘ **$**
string-item-declaration ::= **STRING** ∘ name$_{\text{of-item}}$ ∘ description ∘ number$_{\text{word-index}}$ ∘ number$_{\text{bit-index}}$ ∘ ⟨null | packing mode⟩ ∘ number$_{\text{frequency-of-occurrence}}$ ∘ number$_{\text{of-items-per-word}}$ ∘ **$**

Examples:

```
ITEM KEY A 6 U 3 12 $
STRING BEAD H 2 4 00 D 1 3 $
```

Several elements must be added to complete the description of items for specified entry-structure tables. Such descriptions also contain a word-index number and a bit-index number. Together, these indicate the origin, in terms of word within entry and bit within word,[10] of the storage

[8] Many computers have instructions that, by effectively partitioning memory registers into two or more segments, greatly facilitate extracting values from or inserting them into these "sub-register" segments.

[9] Names so constructed are subject to the normal requirements for uniqueness.

[10] Word 0 bit 0 indicates the first bit position in the entry.

cell containing the item. Packing mode may be included in such an item description to indicate whether this storage cell consists of one or more registers (no packing), of one or more subregisters (medium packing), or of one or more bit positions (dense packing).

A string item occurs in a specified entry-structure table not just once but many times per entry. The number of such occurrences may vary from entry to entry. (Keeping track of this variation is most commonly done by declaring and maintaining a control item, as part of the entry, in which a count of the number of occurrences is kept.) Two additional elements are appended to the description of string items. The first indicates frequency of occurrence in terms of the number of words to the next occurrence of the string item in the entry.[11] The second indicates the number of occurrences of the string item per word.

In designating the value of any particular string item in a specified entry-structure table, a 2-component index list must be appended, as a subscript, to the string item's name. The first component distinguishes the item within the entry (starting with a 0 index), and the second distinguishes the entry itself. Structured item declarations and string item declarations are incorporated into structured entry lists and thus into specified entry-structure table declarations.

structured-entry-list ::= ⟨structured-item-declaration | structured-item-declaration ∘ **BEGIN** ∘ 1-dimensional-constant-list ∘ **END** | string-item-declaration | string-item-declaration ∘ **BEGIN** ∘ 2-dimensional-constant-list ∘ **END**⟩ ∘ ⟨null | structured-entry-list⟩

specified-entry-structure-table-declaration ::= **TABLE** ∘ ⟨null | name$_{\text{of-table}}$⟩ ∘ table-type ∘ number$_{\text{of-entries}}$ ∘ ⟨null | entry-type⟩ ∘ number$_{\text{of-words-per-entry}}$ ∘ **$** ∘ **BEGIN** ∘ structured-entry-list ∘ **END**

Example, which declares a table each of whose entries associates a topic-phrase with a variable number of document references:

```
TABLE   AUTOMATIC'INDEX V 20000 S 1 $
          BEGIN
ITEM    TOPIC'PHRASE H 33 0 00 D $
ITEM    NUMBER'OF'REFERENCES A 12 U 5 24 D $
STRING  DOCUMENT'REFERENCE A 18 U 6 00 D 1 2 $
          END
```

Assuming a 36-bit word-size, **TOPIC'PHRASE** starts in word 0 bit 00 of each entry; **NUMBER'OF'REFERENCES** starts in word 5 bit 24; and the first **DOCUMENT'REFERENCE** starts in word 6 bit 00, appearing there and in each word thereafter, twice per word—to the number of references specified.

For fixed-length entries, the number of words per entry in a specified entry-structure table declaration simply indicates entry length. For variable-length entries, however, it must indicate some common divisor of all the different possible entry lengths. (Often, this is most conveniently one.) In either case, the indicated number of words per entry is used as a factor both in allocating and in indexing the table.[12]

The other elements in a specified entry-structure table declaration have the same meaning they would have in an ordinary table declaration. Serial entry-type must be indicated for a table with variable-length entries, though. Notice that overlay declarations are not needed in specified entry-structure table declarations, since storage allocation within the entry is explicitly indicated.

9. Functional Modifiers

Functional modifiers are, in a sense, extensions to basic JOVIAL, which is essentially an item-manipulating language. They facilitate the manipulation both of larger data elements (i.e., entries and tables) and of smaller data elements (i.e., segments of the machine-language symbols representing item values).

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

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

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

Example, which records the addition of a new entry to the (variable-length) **PAYROLL** table:

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

For variable-length tables, **NENT** serves as a counter that must be maintained by the program itself whenever it changes the table's length. (Initially, the value designated by **NENT** for a variable-length table is undefined.) For rigid-length tables, on the other hand, **NENT** serves as a preset compilation parameter in denoting table-length.

9.2 THE **NWDSEN** MODIFIER. Another parameter in table processing is the amount of storage allocated to a table entry (and thus to the entire table). This unsigned, integral value, which is constant throughout the execution of the program, is expressed in number of words (or registers) per entry.

numeric-formula ::= **N$_{\text{umber-of}}$ W$_{\text{or}}$DS$_{\text{per}}$ EN$_{\text{try}}$** ∘ **(** ∘ name$_{\text{of-table-or table-item}}$ ∘ **)**

Example:

```
PAYROLL'LENGTH = NWDSEN (PAYROLL) * NENT (PAYROLL) $
```

Though ordinarily seldom used, **NWDSEN** is needed in executive programs that do dynamic storage allocation.

9.3 THE **ALL** MODIFIER. A very common loop in JOVIAL programming cycles through an entire table, processing

[11] Where a frequency of occurrence k is indicated, every kth word after the first occurrence of the string item contains similarly allocated occurrences.

[12] In a Serial entry-type, specified entry-structure table with i entries and j words per entry, entry k refers to the entry beginning in word $j*k$ of the $(i*j)$-word table. In a similar, Parallel entry-type table, entry k begins in word k and continues in words $k + i$, $k + (2*i)$, $k + (3*i)$, etc.

one entry each pass. Such a loop can be created with a for-clause that uses the **ALL** modifier.

for-clause ::= **FOR** ∘ loop-counter ∘ **=** ∘ **ALL** ∘ **(** ∘ name$_{\text{of-table-or-table-item}}$ ∘ **)** ∘ **$**

Example:

```
FOR T = ALL (PAYROLL) $
```

The use of a for-clause containing the **ALL** modifier creates a loop with an undefined direction of processing; that is, entry 0 is processed either on the first or on the last pass through the loop.

9.4 The **ENTRY** Modifier. As mentioned before, a table entry is a conglomeration of related items. The **ENTRY** modifier allows an entry to be treated as a single value, represented by a single, composite[13] machine-language symbol.

entry-variable ::= **ENTRY** ∘ **(** ∘ name$_{\text{of-table-or-table-item}}$ ∘ **($** ∘ index$_{\text{of-entry}}$ ∘ **$)** ∘ **)**

entry-formula ::= **0** | entry-variable

boolean-formula ::= entry-variable ∘ ⟨**EQ** | **NQ**⟩ ∘ entry-formula

assignment-statement ::= entry-variable ∘ **=** ∘ entry-formula ∘ **$**

exchange-statement ::= entry-variable ∘ **==** ∘ entry-variable ∘ **$**

Example, which eliminates "empty" entries from the **PAYROLL** table:

```
FOR I = ALL (PAYROLL) $
  BEGIN SEEK'EMPTY.
  IF ENTRY (PAYROLL($I$)) EQ 0 $
    BEGIN
    NENT (PAYROLL) = NENT (PAYROLL) — 1 $
    IF I LS NENT (PAYROLL) $
      BEGIN
      ENTRY (PAYROLL($I$)) == ENTRY (PAYROLL
        ($NENT(PAYROLL)$)) $
      GOTO SEEK'EMPTY $
  END END END
```

The index subscripting the table or table-item name distinguishes the entry from others in the table. An entry's value may be denoted by **0** if all its items have values represented by zero; otherwise its value is not denotable. Comparing (for equality or inequality), assigning, and exchanging of entry values all operate as if on unsigned integers.[14]

9.5 The **BIT** and **BYTE** Modifiers. The machine-language symbol representing any item's value may be considered a string of bits; or, in the case of literal items, of 6-bit bytes (or characters). In either case, both bits and bytes are indexed from left to right, starting with 0.

numeric-variable ::= **BIT** ∘ **($** ∘ index$_{\text{of-first-bit}}$ ∘ ⟨null | **,** ∘ index$_{\text{of-number-of-bits}}$⟩ ∘ **$)** ∘ **(** ∘ name$_{\text{of-item}}$ ∘ ⟨null | subscript⟩ ∘ **)**

literal-variable ::= **BYTE** ∘ **($** ∘ index$_{\text{of-first-byte}}$ ∘ ⟨null | **,** ∘ index$_{\text{of-number-of-bytes}}$⟩ ∘ **$)** ∘ **(** ∘ name$_{\text{of-literal-item}}$ ∘ ⟨null | subscript⟩ ∘ **)**

Example:

```
CONVERT'CARD'IMAGE. "A ROUTINE TO CONVERT
FROM A PUNCHED CARD IMAGE TO AN 80-CHARACTER,
HOLLERITH-CODED, LITERAL VALUE. ILLEGAL 'UNCH
COMBINATIONS ARE NOT REJECTED AND MAY CAUSE
SPURIOUS RESULTS."
BEGIN
ARRAY PUNCH 12 80 B $ ITEM CARD H 80 $ ITEM COLUMN
  H 1 $
FOR J = 0,1,79 $
  BEGIN
  COLUMN = O(00) $
  FOR I = 0,1,11 $
    BEGIN
    IF PUNCH($I,J$) $
      BEGIN
      IF I LQ 2 $ BIT($0,6$)(COLUMN) = BIT($0,6$)
        (COLUMN) + (I+1) * O(20) $
      IF I GR 2 $ BIT($0,6$)(COLUMN) = BIT($0,6$)
        (COLUMN) + (I—2) $
    END  END
  BYTE($J$)(CARD) = COLUMN $
END END
```

The **BIT** modifier allows any segment of the bit-string representing the value of any item to be designated as an unsigned, integral variable. Similarly, the **BYTE** modifier allows any segment of the byte-string representing the value of any literal item to be designated as a literal variable. The first bit or byte of the segment and the number of bits or bytes in the segment are specified by the 2-component index list appended to the modifier. If a one-bit or one-byte segment is desired, the second component of the index list, specifying number of bits or bytes, may be omitted.

9.6 The **MANT** and **CHAR** Modifiers. A floating-point machine-language symbol representing a numeric value consists of: a mantissa, which is a signed fraction representing the significant digits of the value; and a characteristic, which is a signed integer representing the base two exponent of an implicit scaling factor for the mantissa. Either component of any floating-point item can be designated as a fixed-point variable.

numeric-variable ::= ⟨**MANT**$_{\text{issa}}$ | **CHAR**$_{\text{acteristic}}$⟩ ∘ **(** ∘ name$_{\text{of-floating-point-item}}$ ∘ ⟨null | subscript⟩ ∘ **)**

Example, which specifies the fixed-point value of the floating-point item, **BETA**:

```
MANT (BETA) * 2 ** CHAR (BETA)
```

9.7 The **ODD** Modifier. In numeric computations, it is occasionally necessary to determine whether the least significant bit of the machine-language symbol representing the value of a loop counter or of a numeric item represents a magnitude of one, or of zero—for integers, in other words, whether the value is odd or even.[15]

boolean-variable ::= **ODD** ∘ **(** ∘ loop-counter ∘ **)** | **ODD** ∘ **(** ∘ name$_{\text{of-floating-or-fixed-point-item}}$ ∘ ⟨null | subscript⟩ ∘ **)**

Examples:

```
ODD(I)
ODD(GAMMA($X,Y,Z$))
```

With the **ODD** modifier, the least significant bit of any loop counter or floating- or fixed-point item can be designated as a Boolean variable: True if it represents a magnitude of one; False if it represents a magnitude of zero.

[13] Composite as stored, not as declared, since item overlays and unused cells are not unscrambled.

[14] When different-size entries are involved, then, the shorter is (in effect) prefixed by registers containing zero.

[15] **ODD** is somewhat of a misnomer when applied to non-integral values.

10. Miscellaneous Declarations and Statements

10.1 Initial Value Declarations. It is often necessary to declare items with specific initial values, for use as: preset parameters, arrays and tables of constants, or initial data. The initial value of a simple item may be denoted, within either an item declaration or a mode declaration, by a single constant, which must denote a value assignable to the item. This constant may be inserted after the item description and the **P** for *P*reset descriptor, or it may replace these entirely for numeric and literal values. An array item, table item, or string item, on the other hand, is initialized by a constant list, appended to the declaration. This constant list must correspond both in dimension and assignability to the item it presets.

1-dimensional-constant-list ::= constant ∘ ⟨null | 1-dimensional-constant-list⟩

n-dimensional-constant-list ::= **BEGIN** ∘ $n-1$-dimensional-constant-list ∘ **END** ∘ ⟨null | n-dimensional-constant-list⟩

initial-value-declaration ::= ⟨**ITEM** ∘ name$_{\text{of-item}}$ | **MODE**⟩ ∘ ⟨description ∘ **P**$_{\text{reset}}$ ∘ constant | numeric-constant | literal-constant⟩ ∘ **$** | array-declaration ∘ **BEGIN** ∘ n-dimensional-constant-list ∘ **END** | table-declaration$_{\text{containing-constant-lists}}$

Examples:

```
ITEM ERROR 1.234E—5 $
ITEM READY B P 0 $
MODE A 15 U P O(77777) $
ARRAY LETTER'A 6 5 B $
BEGIN
  BEGIN 0 0 1 0 0 END
  BEGIN 0 1 0 1 0 END
  BEGIN 1 0 0 0 1 END
  BEGIN 1 1 1 1 1 END
  BEGIN 1 0 0 0 1 END
  BEGIN 1 0 0 0 1 END
END
TABLE R 12 $
BEGIN
ITEM MONTH H 3 $ BEGIN 3H(JAN) 3H(FEB) 3H(MAR)
  3H(APR) 3H(MAY) 3H(JUN) 3H(JUL) 3H(AUG) 3H(SEP)
  3H(OCT) 3H(NOV) 3H(DEC) END
ITEM LENGTH A 5 U $ BEGIN 31 28 31 30 31 30 31 31 30 31
  30 31 END
END
```

Adopting the convention that a list of individual constants is a 1-dimensional constant list, an n-dimensional constant list consists of a string of $n-1$-dimensional constant lists, each enclosed in **BEGIN** and **END** brackets. Individual constants in a constant list are associated with individual array, table, and string items as follows: The kth component of an index list subscripting the item name serves to index the elements of a k-dimensional constant list. (To make this rule valid for multi-dimensional arrays, however, the positions of the first and second components of the index list for such arrays must be considered as being reversed—thus: second component, first component, third component, fourth component, etc.[16])

The constants in a constant list must all be the same type and they must, of course, denote values that are assignable to the items being initialized. Partial initialization is possible. A constant list that contains k elements (constants or constant lists) where it could have more will initialize only the first k corresponding items or item sets, leaving the remaining values undefined.

10.2 Overlay Declarations. An overlay declaration, by allocating blocks of storage space, indicates the arrangement, in memory, of previously-declared items, arrays, and tables.

block-list ::= name$_{\text{of-item-or-array-or-table}}$ ∘ ⟨null | , ∘ block-list⟩

overlay-list ::= block-list ∘ ⟨null | = ∘ overlay-list⟩

overlay-declaration ::= **OVERLAY** ∘ overlay-list ∘ **$**

Examples:

```
OVERLAY HEAD, BODY, TAIL $
OVERLAY ALPHA=BETA=GAMMA $
OVERLAY DATE=DAY,MONTH,YEAR $
```

The data elements named in a block list are allocated, in sequence,[17] a block of consecutive units of storage. For arrays and tables, these units are full memory registers. For items, the units are registers, subregisters, or bit positions—depending on whether the item packing mode is No packing, Medium packing, or Dense packing.[18] Each block listed in an overlay declaration is allocated storage beginning at a common, though undefined, origin. Each block thus "overlays" the other blocks listed in the declaration.

A name may appear only once in an overlay declaration, but may appear in other overlay declarations if logical inconsistencies are avoided. To arrange storage allocation for table items, the overlay declaration must appear within the ordinary table declaration. (However, such an overlay declaration may contain only item names previously declared within the same entry list.)

10.3 Define Declarations. A define declaration establishes an equivalence between a name and a string of signs by effectively causing the sign string to be substituted for the name wherever it may subsequently occur as a separate Jovial symbol. This allows the programmer to abbreviate lengthy expressions, to make simple additions to the language, and to create symbolic parameters.

define-declaration ::= **DEFINE** ∘ name ∘ **"** sign-string$_{\text{except-the-"-symbol}}$ **"** ∘ **$**

Examples:

```
DEFINE CARD'SEQUENCE "V(JOKER) V(ACE) V(DEUCE)
  V(TREY) V(FOUR) V(FIVE) V(SIX) V(SEVEN) V(EIGHT)
  V(NINE) V(TEN) V(JACK) V(QUEEN) V(KING)" $
DEFINE UNSIGNED "U" $
DEFINE THE " " $
DEFINE RANK "85" $
```

In using define declarations, several points should be remembered: (1) The sign string being defined should contain at least one sign (which may be a blank) but may not contain a " symbol since this, of course, terminates it.

[16] This rather strange reversal retains the convention of indexing first by row then by column, while allowing rows to be written (as 1-dimensional constant lists) *across* the page.

[17] Except for packed table items, which may be rearranged for storage efficiency.

[18] Outside of ordinary table declarations, "normal" item packing mode is compiler-dependent.

(2) No comments may appear in a define declaration. (3) A defined name should be used only in a context where the sign string it defines will comprise an acceptable JOVIAL expression. (4) Circular definitions are possible and *must* be avoided. (5) A defined name may be redefined at a later point in the program listing, and the latest definition will thereafter be substituted for occurrences of the name.

10.4 STOP STATEMENTS. A stop statement halts or indefinitely delays the sequence of statement executions, and usually indicates an operational end to the program in which it appears. If the program is restarted, execution will resume with the next statement listed unless some other statement is named in the stop statement.

stop-statement ::= **STOP** ∘ **$** | **STOP** ∘ name$_{\text{of-next-statement-to-be executed}}$ ∘ **$**

Examples:

```
STOP $
STOP TASK'4 $
```

10.5 DIRECT-CODE STATEMENTS. A direct-code statement allows the programmer to include a routine coded in a "direct" or machine-oriented programming language among the statements of a JOVIAL program. So that such a routine may manipulate JOVIAL item values, it may include a JOVIAL-like assign statement. Such a statement assigns the value contained in the "accumulator" (an undefined machine register) to be the value designated by an item—or vice versa.

accumulator ::= **A**$_{\text{accumulator}}$ **(** ⟨null | signed number$_{\text{of-fraction-bits}}$⟩ **)**

assign-statement ::= **ASSIGN** ∘ accumulator ∘ **=** ∘ name$_{\text{of-item}}$ ∘ ⟨null | subscript⟩ ∘ **$** | **ASSIGN** ∘ name$_{\text{of-item}}$ ∘ ⟨null | subscript⟩ ∘ **=** ∘ accumulator ∘ **$**

direct-code ::= ⟨sign-string$_{\text{except-the-JOVIAL-bracket}}$ | assign-statement⟩ ∘ ⟨null | direct-code⟩

direct-code-statement ::= **DIRECT** ∘ direct-code ∘ **JOVIAL**

The accumulator is designated by the letter **A** followed by a parenthesized number indicating the number of fractional bit positions within the register—usually zero for all but fixed-point numeric values. If the integer is omitted, the register contains a floating-point numeric value.

The effect of a direct-code statement, being machine dependent, is undefined.

10.6 ALTERNATIVE STATEMENTS. A Boolean formula and the associated simple or compound statement following it together constitute an alternative. An alternative statement, consisting mainly of a string of alternatives separated by the **ORIF** symbol, selects for execution the one such statement associated with the first True Boolean formula in the string, if any. The effect of an alternative statement is therefore equivalent to that of the selected statement by itself.[19]

alternative ::= boolean-formula ∘ **$** ∘ ⟨simple-statement | compound-statement⟩

alternative-list ::= **ORIF** ∘ alternative ∘ ⟨null | alternative-list⟩ | name ∘ **.** ∘ alternative-list

alternative-statement ::= **IFEITH**$_{\text{er}}$ ∘ alternative ∘ alternative-list ∘ **END**

[19] Ignoring function side-effects.

Example:

```
IFEITH NUMBER EQ 0 $ SIGN = 0 $
ORIF NUMBER GR 0 $ SIGN = +1 $
ORIF 1 $ SIGN = —1 $
END
```

Statement names can be inserted in the string of alternatives so that an appropriate go-to statement can skip the initial ones.

10.7 CLOSED STATEMENTS. A closed statement is a closed and parameterless subroutine whose execution may only be correctly invoked by an appropriate go-to statement. The normal successor to a closed statement is the statement listed after the invoking go-to statement.

closed-statement ::= **CLOSE** ∘ name$_{\text{of-closed-statement}}$ ∘ **$** ∘ ⟨simple-statement | compound-statement⟩

Example:

```
CLOSE SHELL'SORT $ "A CLOSED STATEMENT WHICH
SORTS A TABLE'S ENTRIES BY KEY ITEM, USING
SHELL'S SORTING ALGORITHM AS DESCRIBED IN ACM
COMMUNICATIONS, JULY 1959."
BEGIN
DEFINE KEY "name of table item" $ "TO BE FILLED IN BY
THE USER BEFORE COMPILATION."
IF NENT (KEY) GR 1 $
  BEGIN
  FOR M = NENT (KEY)/2, —(M + 1), 1 $
    BEGIN
    FOR J = 1, 1, NENT (KEY) — M $
      BEGIN
      FOR I = J—1, —M, 0 $
        BEGIN
        IF KEY($I$) GR KEY($I+M$) $ ENTRY (KEY($I$))
          == ENTRY (KEY($I+M$)) $
END END END END END
```

In using any closed statement, the programmer must see that it is entered only by a go-to statement referring to it by name (e.g., **GOTO SHELL'SORT $**,) never by the name of any of the statements within it, and never as part of the normal, listed sequence of statement executions. Furthermore, while a closed statement may call other closed statements, it may not call itself—either directly or indirectly.

10.8 RETURN STATEMENTS. A return statement indicates a transfer of control to the implicit exit routine that is automatically inserted after the last listed statement within a closed statement or a procedure. A return statement may therefore appear only within a closed statement or a procedure.

return-statement ::= **RETURN** ∘ **$**

Example:

```
RETURN $
```

11. Procedures

A procedure is a self-contained subroutine with a fixed and ordered set of parameters. A procedure is permanently defined by a procedure declaration and invoked either by a procedure statement or by a function.

The actual parameters of a procedure statement or a function are either (1) values, as specified by input formulas or designated by output variables, or (2) names,

indicating arrays, tables, or statements. The formal parameters of a procedure declaration are "dummy" names, corresponding to the actual parameters of the procedure statement or the function.

actual-input-parameter-list ::= ⟨formula | name$_{\text{of-array-or-table}}$⟩ ∘ ⟨null | , ∘ actual-input-parameter-list⟩
formal-input-parameter-list ::= name ∘ ⟨null | , ∘ formal-input-parameter-list⟩
actual-output-parameter-list ::= ⟨variable | name$_{\text{of-array-or-table}}$ | name$_{\text{of-statement}}$ ∘ .⟩ ∘ ⟨null | , ∘ actual-output-parameter-list⟩
formal-output-parameter-list ::= ⟨name | name ∘ .⟩ ∘ ⟨null | , ∘ formal-output-parameter-list⟩

The actual parameters of a procedure statement or a function must correspond to formal parameters of the procedure declaration both in number and in sequence. (Actual parameters may not, therefore, be omitted.) In addition, an actual parameter must agree with its corresponding formal parameter—in data type (i.e., numeric, literal, status, or Boolean) for "value" parameters, and in grammatical usage for "name" parameters. Note that an output parameter which is or which corresponds to a statement name, must be followed by the . (period) separator.

11.1 PROCEDURE DECLARATIONS. A procedure declaration consists of: a heading, which declares the procedure's name and lists its formal parameters, if any; a list of declarations, which describe the information environment peculiar to the procedure, if any; and a compound statement, which forms the body of the procedure.

declaration-list ::= declaration ∘ ⟨null | declaration-list⟩
formal-parameter-list ::= formal-input-parameter-list | = ∘ formal-output-parameter-list | formal-input-parameter-list ∘ = ∘ formal-output-parameter-list
procedure-declaration ::= **PROC**$_{\text{edure}}$ ∘ name$_{\text{of-procedure}}$ ∘ ⟨null | (∘ formal-parameter-list ∘)⟩ ∘ $ ∘ ⟨null | declaration-list⟩ ∘ compound-statement

Examples:

```
PROC SET'DIAGONAL (VALUE=MATRIX,NONE.) $
"WHICH ASSIGNS THE INPUT VALUE TO THOSE ITEMS
ON THE MAIN DIAGONAL OF ANY 50 X 50 FLOATING-
POINT MATRIX THAT ARE ROUGHLY EQUAL TO THE
INPUT VALUE. IF NO ASSIGNMENTS ARE MADE, THE
PROCEDURE EXITS TO 'NONE'."
ITEM VALUE F R $
ARRAY MATRIX 50 50 F R $
ITEM NO'ASSIGNMENT B $
BEGIN
NO'ASSIGNMENT = 1 $
FOR I = 0, 1, 49 $
  BEGIN
  IF CHAR (VALUE) EQ CHAR (MATRIX($I,I$)) $
    BEGIN
    MATRIX($I,I$) = VALUE $
    NO'ASSIGNMENT = 0 $
  END END
IF NO'ASSIGNMENT $ GOTO NONE $
END
PROC RANDOM $ "FUNCTION. MULTIPLICATIVE
  PSEUDO-RANDOM NUMBER GENERATOR."
ITEM RANDOM A 48 U P 5391821890627261 $
ITEM TEMPORARY A 96 U $
BEGIN
TEMPORARY = RANDOM * RANDOM $
RANDOM = BIT($24,48$)(TEMPORARY) $
END
```

Those formal parameters corresponding to actual, "value" parameters must be declared as (simple) items in the declaration list. Those corresponding to arrays or tables must be declared in the declaration list as arrays or tables, to provide the procedure with a fixed definition of their structure, as only their storage locations are transmitted to the procedure.

Names declared inside a procedure, both formal parameters and otherwise, are defined for the procedure only. They bear no relation to identical names used outside the procedure—though outside names can, of course, be used inside procedures.

JOVIAL procedures may invoke other procedures, either through functions or through procedure statements. However, they may not invoke themselves, either directly or indirectly.

In order for a procedure to specify a function value, the procedure name itself must be considered the sole, formal output parameter. It must be declared as an item in the procedure's declaration list and it should be assigned the function value during the execution of the procedure.

11.2 PROCEDURE STATEMENTS. To execute the process defined in a procedure declaration, it is necessary to invoke the procedure by a procedure statement (or by a function). A procedure statement, which may be thought of as an abbreviated description of the process it invokes, has a format similar to that of the heading part of a procedure declaration.

actual-parameter-list ::= actual-input-parameter-list | = ∘ actual-output-parameter-list | actual-input-parameter-list ∘ = ∘ actual-output-parameter-list
procedure-statement ::= name$_{\text{of-procedure}}$ ∘ ⟨null | (∘ actual-parameter-list ∘)⟩ ∘ $

Examples:

```
SET'DIAGONAL (0. = ALPHA, ERROR.) $
COMPLEX'ADD (REAL($I$)+1,IMAG($I$),BETA**2,
  BETA=REAL($I$),IMAG($I$)) $
```

The procedure is executed as if its formal parameters either designated actual-parameter values, or were replaced by actual-parameter names. To effect this, formal input-parameter items are assigned corresponding actual input-parameter values prior to the execution of the procedure, and formal output-parameter item values are assigned to corresponding actual output-parameter variables after this execution. (For this purpose, the execution of a procedure is considered terminated only by the execution of the last statement listed in the procedure declaration, by the execution of a return statement, or by the execution of a go-to statement containing a formal output parameter that denotes a statement name.)

12. Switches

A switch is a routine for computing a statement name and, thus, for deciding among several alternate sequences of operation. It is permanently defined by a switch declaration and invoked by a switch call. A switch declaration consists primarily of a switch list, whose major elements

are statement names or switch calls. The routine invoked by a switch call selects one of these elements, thus determining (perhaps by another switch) the "value" of the switch.

JOVIAL has two kinds of switches: the index switch, whose value is determined by an index specified in the switch call; and the item switch whose value is determined by the value of an item named in the switch declaration.

index-switch-list ::= ⟨null | name$_{\text{of-statement}}$ | switch-call⟩ ∘ ⟨null | , ∘ index-switch-list⟩

item-switch-list ::= constant ∘ = ∘ ⟨name$_{\text{of-statement}}$ | switch-call⟩ ∘ ⟨null | , ∘ item-switch-list⟩

switch-declaration ::= **SWITCH** ∘ name$_{\text{of-index-switch}}$ ∘ = ∘ (∘ index-switch-list ∘) ∘ $ | **SWITCH** ∘ name$_{\text{of-item-switch}}$ ∘ (∘ name$_{\text{of-item-or-file}}$ ∘) ∘ = ∘ (∘ item-switch-list ∘) ∘ $

switch-call ::= name$_{\text{of-index-switch}}$ ∘ ($ ∘ index$_{\text{for-switch-list}}$ ∘ $) | name$_{\text{of-item-switch}}$ ∘ ⟨null | subscript$_{\text{for-switch-item}}$⟩

Examples:

```
SWITCH GET'RATE = (GET'RATE($DAY$),
  SUNDAY'RATE,WEEKDAY'RATE,WEEKDAY'RATE,
  WEEKDAY'RATE,WEEKDAY'RATE,WEEKDAY'RATE,
  SATURDAY'RATE) $
GOTO GET'RATE($1$) $
```

The index in an index-switch call selects one of the n positions in the switch list of the corresponding switch declaration. (This index may therefore range from 0 through $n-1$ only.) Any of the positions in an index-switch list may be empty (null) thus effectively specifying the first statement listed after the switch-invoking go-to statement.

The item name given in an item-switch declaration and the index list (if any) subscripting the switch name in the call together designate an item value. This value selects from the item-switch list in the declaration the statement name or switch call paired with the first constant in the list that denotes a value equal to it. If no such constant is listed in the declaration, then the switch effectively specifies the first statement listed after the switch-invoking go-to statement. (The use of a file name in an item switch is treated in Section 13.2, Testing and Positioning Files.)

A switch is invoked, directly or indirectly, by a go-to statement and may compute the name of a closed statement whose normal successor would be the first statement listed after the go-to statement.

13. Input-Output and Files

Many data storage devices impose accessing restrictions in that storing or loading an individual value may, for efficiency, ordinarily involve the transfer of an entire block of data. Such devices are termed *external* storage devices, as contrasted with the *internal* memory of the computer. To allow a reasonably efficient description of input-output processes, therefore, all data entering or leaving the computer's internal memory is organized into files. A file is thus a body of data contained in some external storage device, such as punched cards or tape, or magnetic tape, discs, or drums.

13.1 FILE DECLARATIONS. A file is a list of records, which are themselves strings of bits or of 6-bit, Hollerith-coded bytes. A file's records are either all binary or all Hollerith, and they are generally homogeneous in size, content, and format. (When heterogeneous records are organized into a file, the program must provide for distinguishing among them.) Record format is not described in the file declaration, however, since it is determined by the input or output records in the statements that read or write the file.

file-declaration ::= **FILE** ∘ name$_{\text{of-file}}$ ∘ ⟨**B**$_{\text{inary}}$ | **H**$_{\text{ollerith}}$⟩ ∘ number$_{\text{of-records}}$ ∘ ⟨**V**$_{\text{ariable-record-length}}$ | **R**$_{\text{igid-record-length}}$⟩ ∘ number$_{\text{of-bits-or-bytes-per-record}}$ ∘ status-list ∘ name$_{\text{of-storage-device}}$ ∘ $

Examples:

```
FILE INVENTORY B 10000 V 480 V(UNREADY) V(READY)
  V(BUSY) V(ERROR) TAPEDRIVE'A $
FILE LINE'OF'PRINT H 500 R 120 V(UNREADY) V(READY)
  LINE'PRINTER $
```

In the file declaration, both number of records and number of bits or bytes per record may be estimated maximums. The listed status constants are associated with the file name and denote the possible states of the storage device containing the file.[20] The storage-device name indicates, in compiler-dependent terms,[21] the particular storage device containing the file.

13.2 TESTING AND POSITIONING FILES. The status of a file is denoted by one of the status constants listed in the declaration. These are associated with the file name, which may be considered as a status item that is automatically updated prior to any comparison according to the current state of the storage device containing the file. File status may thus be tested with a relational Boolean formula, or by means of a call to an appropriate *item* switch.

Examples:

```
SWITCH CHECK'INVENTORY'FILE (INVENTORY) =
  (V(UNREADY)=PROCESS'FILE'END,V(BUSY)=WAIT,
  V(ERROR)=PROCESS'ERROR) $
CLOSE WAIT $ BEGIN STEP1. IF INVENTORY EQ
  V(BUSY) $ GOTO STEP1 $ STEP2. GOTO
  CHECK'INVENTORY'FILE $ END
```

A JOVIAL file is a self-indexing storage device, meaning that the record available for transfer to or from the file depends on the file's current position. The records of an n-record file are indexed from 0 through $n-1$, and the index of the record currently available for transfer is designated, as file-position, with the functional modifier **POS**, operating on the name of an active[22] file. File position ranges from 0 (indicating "rewound") thru n (indicating "end-of-file"). The transfer of a record to or from a file automatically increments the file position by one. Furthermore, where the storage device allows, file position is a

[20] Storage-device names and the number and meaning of their possible states are compiler-dependent, so that anyone wishing to declare a workable file must refer to the pertinent documentation for a particular JOVIAL compiler.

[21] This name may indicate such things as: drum or disk address; file-index for multi-file tapes; etc.

[22] An active file is one that has been "activated" by the execution of an open-input or an open-output statement, as described in the following sections.

variable that may be altered by the assignment of an arbitrary numeric value. The file is then called an *addressable* file, as opposed to a *serial* file, where such a general positioning operation is to be avoided as impossible or inefficient.

numeric-variable ::= **POS**$_{\text{ition}}$ ∘ **(** ∘ name$_{\text{of-active-file}}$ ∘ **)**

Example:

```
POS (INVENTORY) = 0 $
```

13.3 INPUT STATEMENTS. A file may be read, one record at a time, by the execution of a series of input statements. The first statement executed in such a series must be an open-input statement, which activates the file, and the last must be a shut-input statement, which deactivates it. Input records so read may be variables, entire arrays, entire tables, sequences of table entries, or individual table entries.

input-record ::= variable | name$_{\text{of-array}}$ | name$_{\text{of-table}}$ | name$_{\text{of-table}}$ ∘ **($** ∘ index$_{\text{of-first-entry}}$ ∘ ... ∘ index$_{\text{of-last-entry}}$ ∘ **$)** | name$_{\text{of-table}}$ ∘ **($** ∘ index$_{\text{of-entry}}$ ∘ **$)**

input-statement ::= **OPEN** ∘ **INPUT** ∘ name$_{\text{of-inactive-file}}$ ∘ ⟨null | input-record⟩ ∘ **$** | **INPUT** ∘ name$_{\text{of-active-file}}$ ∘ input-record ∘ **$** | **SHUT** ∘ **INPUT** ∘ name$_{\text{of-active-file}}$ ∘ ⟨null | input-record⟩ ∘ **$**

Example:

```
BEGIN
OPEN INPUT INVENTORY $
PROCESS'INVENTORY. INPUT INVENTORY ARTICLE $
GOTO CHECK'INVENTORY'FILE $
GOTO PROCESS'ARTICLE $
GOTO PROCESS'INVENTORY $
PROCESS'FILE'END. SHUT INPUT INVENTORY $
END
```

A read operation transfers the string of bits or bytes comprising a file record from the file into the computer's internal memory, to represent the value or values of a designated input record. A read is terminated when the entire input record has been represented. (Any bits or bytes left in the file record go unread and are skipped over.) A read is also terminated when the string of bits or bytes of the file record is exhausted. (The remainder of the input record, if any, is undefined.)

An open-input statement activates an inactive file and initializes its position to zero. A shut-input statement deactivates an active file. Any input statement that designates an input record initiates a read operation that will transfer a record from the file into the computer's internal memory, thus incrementing file-position by one.

If the (compiler-dependent) file characteristics permit,[23] an input or shut-input statement may involve a file activated by an open-output statement.

13.4 OUTPUT STATEMENTS. A file may be written, one record at a time, by the execution of a series of output statements. The first statement executed in such a series must be an open-output statement, which activates the file, and the last must be a shut-output statement, which deactivates it. Output records so written may be numeric or literal constants, variables, arrays, etc.

output-record ::= numeric-constant | literal-constant | input-record

output-statement ::= **OPEN** ∘ **OUTPUT** ∘ name$_{\text{of-inactive-file}}$ ∘ ⟨null | output-record⟩ ∘ **$** | **OUTPUT** ∘ name$_{\text{of-active-file}}$ ∘ output-record ∘ **$** | **SHUT** ∘ **OUTPUT** ∘ name$_{\text{of-active-file}}$ ∘ ⟨null | output-record⟩ ∘ **$**

Examples:

```
OPEN OUTPUT PERSONNEL'FILE $
OUTPUT PERSONNEL'FILE EMPLOYEE'RECORD
  ($I...I+50$) $
SHUT OUTPUT PERSONNEL'FILE EMPLOYEE'RECORD
  ($I...NENT(EMPLOYEE'RECORD)—1$) $
```

A write operation transfers the string of bits or bytes representing a designated output record from the computer's internal memory out onto the file, as a file record. A write is terminated when the entire output record has been transferred. (For rigid record-length files, the remainder of the file record, if any, is undefined.) A write is also terminated when the number of bits or bytes transferred equals the declared maximum file-record size.

An open-output statement activates an inactive file and initializes its position to zero. A shut-output statement deactivates an active file. Any output statement that designates an output record initiates a write operation that will transfer a record from the computer's internal memory out onto the file, thus incrementing file-position by one.

If the (compiler-dependent) file characteristics permit,[24] an output or shut-output statement may involve a file activated by an open-input statement.

14. Programs

A JOVIAL program is a list of declarations and statements enclosed in the **START** and **TERM** brackets. If a statement name is not provided after the **TERM**, the first statement in the program's execution sequence is the first statement listed that is not part of a procedure declaration. And if this first-listed statement is named, its name can also be considered as the name of the program. The **$** separator indicates the typographic end of the program.

program ::= **START** ∘ ⟨null | declaration-list⟩ ∘ ⟨null | name$_{\text{of-program}}$ ∘ **.**⟩ ∘ statement-list ∘ **TERM** ∘ ⟨null | name$_{\text{of-first-statement-to-be-executed}}$⟩ ∘ **$**

REFERENCES

1. Preliminary report—international algebraic language. *Comm. ACM 1*, 12 (1958).
2. ENGLUND AND CLARK. The Clip translator. *Comm. ACM 4* (Jan. 1961).
3. Revised report on ALGOL 60. *Comm. ACM 5* (Jan. 1963).

[23] Some files are write-only in type.

[24] Some files are read-only in type.

INDEX OF SYMBOLS AND TERMS

The following is an index of the JOVIAL symbols and metalinguistic terms appearing in the syntactic formulas of this report. Section numbers in parentheses refer to sections that contain term-defining formulas; section numbers not in parentheses refer to sections that contain symbol- or term-using formulas. (To distinguish between symbols and terms, notice that "symbol" is a term, whereas "**TERM**" is a symbol.)

accumulator (10.5) 10.5
actual-input-parameter-list (11) 5.3.1, 11, 11.2
actual-output-parameter-list (11) 11, 11.2
actual-parameter-list (11.2) 11.2
ALL 3.1, 9.3
alternative (10.6) 10.6
alternative-list (10.6) 10.6
alternative-statement (10.6) 6.4
AND 3.1, 5.3.4
Arithmetic 3.1, 5.1
arithmetic-operator (3.1) 3.1, 5.3.2
ARRAY 3.1, 6.3
array-declaration (6.3) 6, 10.1
ASSIGN 10.5
assignment-statement (6.6, 9.4) 6.4
assign-statement (10.5) 10.5

BEGIN 3.1, 6.4, 7.1, 8.1, 8.3, 10.1
Binary 3.1, 13.1
BIT 3.1, 9.5
blank 3, 5
block-list (10.2) 10.2
Boolean 3.1, 5.1
Boolean-constant (3.3) 3.3, 5.3.4
Boolean-formula (5.3.4, 9.4) 5.3, 5.3.4, 6.8, 10.6
Boolean-item-description (5.1) 5.1
Boolean-variable (5.2, 9.7) 5.2, 5.3.4
bracket (3.1) 3.1
BYTE 3.1, 9.5

CHARacteristic 3.1, 9.6
CLOSE 3.1, 10.7
closed-statement (10.7) 6.4
comment (4) 5
complex-statement (6.4, 6.5) 6.4, 6.5
compound-statement (6.4, 6.5) 6.4, 6.5, 6.8, 7.1, 10.6, 10.7, 11.1
conditional-statement (6.8) 6.4
constant (3.3) 3, 10.1, 12

declaration (6) 6.4, 11.1
declaration-list (11.1) 11.1, 14
declarator (3.1) 3.1
DEFINE 3.1, 10.3
define-declaration (10.3) 6
delimiter (3.1) 3
Dense 3.1, 8.1
description (5.1) 6.1, 6.2, 6.3, 8.3, 10.1
descriptor (3.1) 3.1
dimension-list (6.3) 6.3
DIRECT 3.1, 10.5
direct-code (10.5) 10.5
direct-code-statement (10.5) 6.4
Dual 3.1, 5.1
dual-constant (3.3) 3.3, 5.1
dual-item-description (5.1) 5.1

END 3.1, 6.4, 7.1, 8.1, 8.3, 10.1, 10.6
ENTRY 3.1, 9.4
entry-formula (9.4) 9.4
entry-list (8.1) 8.1
entry-type (8.1) 8.1, 8.2, 8.3
entry-variable (9.4) 9.4
EQ 3.1, 9.4
exchange-statement (6.7, 9.4) 6.4

FILE 3.1, 13.1
file-declaration (13.1) 6
file-operator (3.1) 3.1
fixed-constant (3.3) 3.3, 5.1
fixed-point-item-description (5.1) 5.1
Floating 3.1, 5.1
floating-constant (3.3) 3.3, 5.1
floating-point-item-description (5.1) 5.1
FOR 3.1, 7.1, 9.3
formal-input-parameter-list (11) 11, 11.1
formal-output-parameter-list (11) 11, 11.1
formal-parameter-list (11.1) 11.1
formula (5.3) 6.6, 11
for-clause (7.1, 9.3) 7.1
function (5.3.1) 5.3.2, 5.3.3, 5.3.4
functional-modifier (3.1) 3.1

GOTO 3.1, 6.9
go-to-statement (6.9) 6.4
GQ 3.1
GR 3.1

Hollerith 3.1, 5.1, 13.1

identifier (3.2)
IF 3.1, 6.8
IFEITHer 3.1, 10.6
if-clause (6.8) 6.8, 7.1
index (5.2) 5.2, 7.1, 9.4, 9.5, 12, 13.3
index-list (5.2) 5.2
index-switch-list (12) 12
initial-value-declaration (10.1) 6
INPUT 3.1, 13.3
input-record (13.3) 13.3, 13.4
input-statement (13.3) 6.4
integer-constant (3.3) 3.3, 5.1
ITEM 3.1, 6.1, 8.3, 10.1
item-declaration (6.1) 6, 8.1
item-switch-list (12) 12

JOVIAL 3.1, 10.5

letter (3) 3, 3.2, 3.3, 5, 7, 8.2
Like 3.1, 8.2
like-table-declaration (8.2) 8
literal-constant (3.3) 3.3, 5.3.3, 10.1
literal-formula (5.3.3) 5.3, 5.3.4
literal-item-description (5.1) 5.1
literal-relation-list (5.3.4) 5.3.4
literal-variable (5.2, 9.5) 5.2, 5.3.3
logical-operator (3.1) 3.1
loop-counter (3.2) 7, 7.1, 7.2, 9.3, 9.7
loop-indices (7.1) 7.1
loop-statement (7.1) 6.4
LQ 3.1
LS 3.1

MANTissa 3.1, 9.6
mark (3) 3, 5
Medium 3.1, 8.1
MODE 3.1, 6.2, 10.1
mode-declaration (6.2) 6

name (3.2) 3, 3.2, 3.3, 5.2, 5.3.1, 5.3.4, 6.1, 6.3, 6.5, 6.9, 7.1, 8.1, 8.2, 8.3, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 10.1, 10.2, 10.3, 10.4, 10.6, 10.7, 11, 11.1, 11.2, 12, 13.1, 13.2, 13.3, 13.4, 14
NENT 3.1, 9.1
No 3.1, 8.1
NOT 3.1, 5.3.4
NQ 3.1, 9.4
null (2) 3.2, 3.3, 5, 5.1, 5.2, 5.3.1, 5.3.4, 6.3, 6.4, 8.1, 8.2, 8.3, 9.5, 9.6, 9.7, 10.1, 10.2, 10.5, 10.6, 11, 11.1, 11.2, 12, 13.3, 13.4, 14
number (3.3) 3.3, 5.1, 6.3, 8.1, 8.2, 8.3, 10.5, 13.1
numeral (3) 3, 3.2, 3.3, 5, 8.2
numeric-constant (3.3) 3.3, 5.3.2, 10.1, 13.4
numeric-formula (5.3.2, 9.1, 9.2) 5.2, 5.3, 5.3.2, 5.3.4
numeric-item-description (5.1) 5.1
numeric-relation-list (5.3.4) 5.3.4
numeric-variable (5.2, 7, 9.1, 9.6, 13.2) 5.2, 5.3.2
NWDSEN 3.1, 9.2
n-dimensional-constant-list (10.1) 10.1
n−1-dimensional-constant-list (see 10.1) 10.1

o (5) 5, 5.1, 5.2, 5.3.1, 5.3.2, 5.3.4, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 7.1, 7.2, 8.1, 8.2, 8.3, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 11, 11.1, 11.2, 12, 13.1, 13.2, 13.3, 13.4, 14
octal-constant (3.3) 3.3
octal-number (3.3) 3.3
ODD 3.1, 9.7
OPEN 3.1, 13.3, 13.4
OR 3.1, 5.3.4
ordinary-table-declaration (8.1) 8
ORIF 3.1, 10.6
OUTPUT 3.1, 13.4
output-record (13.4) 13.4
output-statement (13.4) 6.4
OVERLAY 3.1, 10.2
overlay-declaration (10.2) 6, 8.1
overlay-list (10.2) 10.2

packing-mode (8.1) 8.1, 8.2, 8.3
Parallel 3.1, 8.1
POSition 3.1, 13.2
Preset 3.1, 10.1
PROCedure 3.1, 11.1
procedure-declaration (11.1) 6
procedure-statement (11.2) 6.4
program (14)

relational-formula (5.3.4) 5.3.4
relational-operator (3.1) 3.1, 5.3.4
RETURN 3.1, 10.8
return-statement (10.8) 6.4
Rigid 3.1, 8.1, 13.1
Rounded 3.1, 5.1

separation (5) 5
separator (3.1) 3.1
sequential-operator (3.1) 3.1
Serial 3.1, 8.1
SHUT 3.1, 13.3, 13.4
sign (3) 3.3, 5
signed (3.3) 3.3, 5.1, 10.5
Signed 3.1, 5.1
sign-string (3.3) 3.3, 4, 10.3, 10.5
simple-statement (6.4, 6.5) 6.4, 6.5, 6.8, 7.1, 10.6, 10.7
specified-entry-structure-table-declaration (8.3) 8
START 3.1, 14
statement (6.4) 6.4
statement-list (6.4) 6.4, 7.1, 14
Status 3.1, 5.1
status-constant (3.3) 3.3, 5.1, 5.3.3
status-formula (5.3.3) 5.3, 5.3.4
status-item-description (5.1) 5.1
status-list (5.1) 5.1, 13.1
status-variable (5.2) 5.2, 5.3.3, 5.3.4
STOP 3.1, 10.4
stop-statement (10.4) 6.4
STRING 3.1, 8.3
string-item-declaration (8.3) 8.3
structured-entry-list (8.3) 8.3
structured-item-declaration (8.3) 8.3
subscript (5.2) 5.2, 9.5, 9.6, 9.7, 10.5, 12
SWITCH 3.1, 12
switch-call (12) 6.9, 12
switch-declaration (12) 6
symbol (3)

TABLE 3.1, 8.1, 8.2, 8.3
table-declaration (8) 6, 10.1
table-type (8.1) 8.1, 8.2, 8.3
TERM 3.1, 14
TEST 3.1, 7.2
test-statement (7.2) 6.4
Transmission-code 3.1, 5.1

Unsigned 3.1, 5.1

Variable 3.1, 8.1, 13.1
variable (5.2) 6.6, 6.7, 11, 13.3

1-dimensional-constant-list (10.1) 8.1, 8.3, 10.1
2-dimensional-constant-list (see 10.1) 8.3
) 3.1, 3.3, 5.3.1, 5.3.2, 5.3.4, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 11.1, 11.2, 12, 13.2
− 3.1, 3.3, 5.3.2
+ 3.1, 3.3, 5.3.2
= 3.1, 6.6, 7.1, 9.4, 10.2, 10.5, 11.1, 11.2, 12
== 3.1, 6.7, 9.4
$ 3.1, 4, 6.1, 6.2, 6.3, 6.6, 6.7, 6.8, 6.9, 7.1, 7.2, 8.1, 8.2, 8.3, 9.3, 9.4, 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 11.1, 11.2, 12, 13.1, 13.4, 14
$) 3.1, 5.2, 9.4, 9.5, 12, 13.3
* 3.1
** 3.1
(3.1, 3.3, 5.3.1, 5.3.2, 5.3.4, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 11.1, 11.2, 12, 13.2
($ 3.1, 5.2, 9.4, 9.5, 12, 13.3
(/ 3.1, 5.3.2
, 3.1, 3.3, 5.2, 7.1, 9.5, 10.2, 11, 12
" 3.1, 4, 10.3
/ 3.1
/) 3.1, 5.3.2
. 3.1, 6.5, 3.3, 7.1, 10.6, 11, 14
... 3.1, 5.1, 13.3