SDC

TM-629

A Programmer's Introduction to Basic JOVIAL

7 August 1961

TECHNICAL MEMORANDUM

(TM Series)

A
Programmer's Introduction
to
Basic JOVIAL

C. J. Shaw

7 August 1961

SYSTEM
DEVELOPMENT
CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA

(Page 2 blank)

A PROGRAMMER'S INTRODUCTION TO BASIC JOVIAL

ABSTRACT

This paper describes the basic elements of JOVIAL, a procedure-oriented and largely computer-independent programming language designed by the System Development Corporation for computer-based command/control systems. JOVIAL is derived from ALGOL 58, with the addition of an input-output notation, a more elaborate data-description capability, and the ability to manipulate symbolic and other non-numeric values.

TABLE OF CONTENTS

PREFACE

Although many optional forms have been provided in JOVIAL to allow the skillful programmer to create an efficient program, this paper is a brief introduction and thus treats only the basic parts of the language, so that a few of the more esoteric options are not described. A complete description of the language may be obtained from the author at the System Development Corporation.

INTRODUCTION

The circuitry of a digital computer allows it to process (input, store, manipulate, and output) information under the direction of a program of instructions supplied by the programmer. Information so processed is composed of basic units called values, which are represented by digitally encoded symbols. A digital computer, therefore, processes not values but binary symbols representing values.

Since a language of binary symbols is convenient only for computers, programmers have developed more intelligible programming languages, composed of alphanumeric symbols, which can usually be automatically translated into machine language by the computer itself. These languages, closely resembling machine languages at first, have evolved in the last few years to become less machine-oriented, more procedure-oriented, and more powerful.

Such a procedure-oriented language is JOVIAL, designed by SDC for programming large, computer-based command/control systems. JOVIAL is a largely computer-independent programming language; compilers for the IBM 709/7090, the CDC 1604, the PHILCO 2000, the AN/FSQ-31, and the AN/FSQ-7 are either operational or being written. JOVIAL is derived from ALGOL 58 (the 1958 version of the International ALGOrithmic Language, as described in the December 1958 issue of the ACM Communications) with the addition of an input/output notation, a more elaborate data-description capability, and the ability to manipulate symbolic and other non-numeric values.

NOTATION

The grammar used to describe JOVIAL syntax consists of rules with the form:

element § string-of-elements

where an element either denotes or exhibits a JOVIAL form. (1) The meta-symbol § signifies syntactic equivalence, while the colon : signifies concatenation and the semicolon ; signifies selection between adjacent elements. (2) A subscript for an element is a semantic cue, with no syntactic effect. (3) The brackets [and] group a string of elements into a single element, while the brackets (and) group a string of elements into a single, optional element. (4) The suffix s signifies a string of one or more of the elements to which it is appended, while an element superscripting it is the separator, if any. (5) A space signifies a string of JOVIAL blanks, and since JOVIAL symbols are normally separated by blanks, this separation convention will not be explicitly indicated.

JOVIAL is a programming language for professional programmers, who are notoriously averse to redundant coding. JOVIAL consequently uses certain abbreviations: S for Signed; F for Floating; B for Boolean; and so on. Indeed, these abbreviations are the normal forms of the language and must be explicitly defined away if the expanded versions are to be used. For example:

```
DEFINE Boolean '' B '' $
```

To simplify the discussion, therefore, it is assumed that definitions like the above have been given. The normal JOVIAL abbreviation is the capitalized, first letter.

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 sign.

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 § Ø;1;2;3;4;5;6;7;8;9

sign § letter;numeral;blank;);(;+;-;*;/;.;,;';=;$

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

Delimiters

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

arithmetic-operator § +;-;*;/;**

relational-operator § EQ;GQ;GR;LQ;LS;NQ

logical-operator § AND;OR;NOT

sequential-operator § IF;GOTO;FOR;TEST;CLOSE;RETURN;STOP;IFEITHer;ORIF

file-operator § OPEN;SHUT;INPUT;OUTPUT

functional-modifier § BIT;BYTE;MANT;CHAR;NENT;ALL;ENTRY;POSition

separator § .;,;=;==;';...;$

bracket § (;);(/;/);($;$);' ';' ';BEGIN;END;DIRECT;JOVIAL;START;TERM

declarator § ITEM;MODE;ARRAY;TABLE;DEFINE;SWITCH;PROCedure;FILE

descriptor § Floating;fixed;Signed;Unsigned;Rounded;Hollerith;Transmission;Status;Boolean;
Variable;Rigid;Preset;Like;Binary

Identifiers

Identifiers name the elements of a JOVIAL program's information environment: statements; switches; procedures; items; tables; and files. Except for context defined statement names, all JOVIAL identifiers must be declared, either in the program, or in a system declaration list (COMPOOL). Identifiers must obviously be distinguishable from delimiters, and from each other, and this can always be done by a unique spelling.

name § letter:[letter;numeral]s(')

Examples

STEPØ1

U2

BRANCH

FLIGHT'POSITION

Constants

A constant denotes a particular data value that is unaffected by program execution. JOVIAL programs manipulate four types of data: numeric (in either floating or fixed-point representation); symbolic or literal (in either computer dependent Hollerith or standard Transmission code representation); status; and Boolean. Numbers, integers, and floating and fixed constants denote numeric values in the conventional, decimal sense; while octal constants have the obvious meaning of octal integers. Literal constants denote JOVIAL sign strings, represented in one of two possible 6-bit-per-sign encodings; status constants are mnemonic names denoting qualities or categories rather than numeric values; and Boolean constants denote either True (by 1) or False (by Ø).

number § numerals

integer § (+;−):number:(E$_{\text{xponent-base-10}}$:number)

floating-constant § (+;-):[number.:(number)];[.:number]:(E$_{xponent-base-10}$:(+;-):number)

fixed-constant § floating-constant:A$_{ffix}$:(+;-):number$_{of-fraction-bits}$

octal-constant § O$_{ctal}$:(:[0;1;2;3;4;5;6;7]s:)

literal-constant § number$_{of-signs}$:H$_{ollerith-code}$;T$_{ransmission-code}$:(:signs:)

status-constant § V$_{alue}$:(:letter;name:)

boolean-constant § 1$_{true}$;0$_{false}$

constant § integer;floating-constant;fixed-constant;literal-constant;status-constant;boolean-constant

Examples

```
0018
-123E4
.5
+0.6789E-6A36
O(777760)
27H(THIS IS A LITERAL CONSTANT.)
11T(SO IS THIS.)
V(EXCELLENT)
1
```

COMMENTS

A comment allows a remark or clarifying text to be included among the symbols of a JOVIAL program, for comments are treated as strings of blanks by the compiler and thus have no operational effect whatever on the program.

blanks § ' ':signs$_{\text{except-the-symbols-' '-and-\$}}$' '

Example

' 'THIS IS A COMMENT.' '

CLAUSES

Strings of JOVIAL symbols, separated by blanks which may be omitted where this does not join a numeral/letter pair, form clauses, which are: item descriptions; variables; and formulas. An item description describes a value; a variable designates a value; and a formula specifies a value.

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 symbol representing it, need be supplied only once, in an item description.

description$_{\text{of-floating-point-item}}$ § Floating (Rounded)

description$_{\text{of-fixed-point-item}}$ § fixed number$_{\text{of-bits}}$ Signed;Unsigned ((+;-): number$_{\text{of-fraction-bits}}$) (Rounded)

description$_{\text{of-literal-item}}$ § Hollerith$_{\text{code}}$;Transmission$_{\text{code}}$ number$_{\text{of-signs}}$

description$_{\text{of-status-item}}$ § Status (number$_{\text{of-bits}}$) status-constants

description$_{\text{of-boolean-item}}$ § Boolean

Examples

```
Floating Rounded

fixed 36 Signed 15

Hollerith $6

Status V(bad) V(poor) V(fair) V(good) V(fine)

Boolean
```

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 principal variables. If an item name designates more than one value, as in an array or table, an index list of numeric formulas enclosed in the subscript brackets ($ and $) distinguishes a particular value.

index § ($ numeric-formulas' $)

variable § name$_{\text{of-item}}$ (index)

Examples

```
alpha

beta($t2$)

gamma($0,t2+13,beta($t2-1$)+1,9*t2$)
```

BIT and BYTE Functional Modifiers

The machine symbol representing any item's value may be considered a string of bits or, in the case of literal items, of 6-bit bytes; indexed left to right in either case from ∅ thru n-1 for an n-element symbol. Although the item is normally the smallest unit of data in JOVIAL, it is occasionally necessary to designate a value represented by part of an item's machine symbol. This function, especially useful with literal values, is performed by the subscripted functional modifiers BIT (which designates an unsigned, integral value) and BYTE (which designates a literal value).

variable$_{\text{of-numeric-type}}$ § BIT ($ numeric-formula$_{\text{index-of-first-bit}}$

〔, numeric-formula$_{\text{number-of-bits}}$〕$) (name$_{\text{of-item}}$ 〔index〕)

variable$_{\text{of-literal-type}}$ § BYTE ($ numeric-formula$_{\text{index-of-first-byte}}$

〔, numeric-formula$_{\text{number-of-bytes}}$〕$) (name$_{\text{of-literal-item}}$ 〔index〕)

Examples

```
BIT($9$)(emp'code($emp$))

BYTE($∅,length$)(message)
```

MANT and CHAR Functional Modifiers

A floating-point machine symbol representing a numeric value consists of: a mantissa or signed fraction, representing the value's significance; and a signed integer characteristic or power-of-two factor for the mantissa, representing the value's scaling. Either component of a simple or subscripted floating-point item may be designated: the mantissa with the functional modifier MANT; and the characteristic with the functional modifier CHAR.

variable$_{\text{of-numeric-type}}$ § MANT;CHAR (name$_{\text{of-floating-point-item}}$ 〔index〕)

Examples

```
CHAR(alpha)

MANT(beta($CHAR(alpha)$))
```

Formulas

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

Functions

A function specifies the value computed by a procedure utilizing the function's calling parameters, that is: values, specified by formulas; and certain environment elements, denoted by name. (Calling parameters are discussed in the section on procedures.)

function § name$_{\text{of-procedure}}$ (⟨[formula;name]s'⟩)

Examples

```
arcsin(2*beta($t2$)**2,#.1E-4A15)

random()

word(symbol'list,1H( ),1H(.),1H(,),18)

state(V(soon),factorial(alpha))

symmetric(matrix'7)
```

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. Where necessary, conversion between fixed and floating-point representation is implied. As in algebra, division by zero is undefined. Fractional or mixed exponents are possible, but since JOVIAL deals only with rational numbers, any exponentiation which would specify a complex root, such as (-2)**.5, is also undefined. Parentheses perform the usual grouping function, as do the absolute magnitude brackets (/ and /).

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

numeric-formula § integer;floating-constant;fixed-constant;octal-constant;

variable$_{\text{of-numeric-type}}$;function$_{\text{of-numeric-type}}$;[(numeric-formula)];

[(/ numeric-formula /)];[(+;-)numeric-formulas[+;-;*;/;**]]

Examples

273

1.889E-6

alpha(0)

log (beta($+2$))

(-273*alpha(0)+(/beta($+2$)/)**-log (beta($+2$)))/1.889E-6

Literal and Status Formulas

Both literal and status formulas specify a value as expressed by a lone operand: a constant; a variable; or a function, of the appropriate type.

literal-formula § literal-constant;octal-constant;variable$_{\text{of-literal-type}}$;

function$_{\text{of-literal-type}}$

status-formula § status-constant;variable$_{\text{of-status-type}}$;function$_{\text{of-status-type}}$

Examples

```
6T(ABACUS)

O(Ø6Ø7Ø61Ø323Ø)

signal

V(cloudy)

weather($airbase$)

type (signal)
```

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. 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 means, is EQual to; NQ means, is uNeQual to; GR means, is GReater than; LQ means, is Less than or eQual to; LS means, is LesS than; and GQ means, 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.

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.

relational-formula § numeric-formulas$_{\text{relational-operator}}$;literal-formulas$_{\text{relational-operator}}$;[variable$_{\text{of-status-type}}$ relational-operator status-formula]

boolean-formula § boolean-constant;variable$_{\text{of-boolean-type}}$;function$_{\text{of-boolean-type}}$;relational-formula;[(boolean-formula)];[[NOT] boolean-formulas$_{[\text{AND;OR}]}$]

formula § numeric-formula;literal-formula;status-formula;boolean-formula

Examples

ø

indicator

legal (signal)

-13 LS alpha LQ +1ØØ 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))

SENTENCES

With certain delimiters, clauses are combined to form statements and declarations, 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.

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 must be formally declared as items before they may be referenced.

declaration § ITEM name$_{\text{of-simple-item}}$ description $

Examples

```
ITEM     p66 Floating $

ITEM     tally fixed 15 Unsigned Rounded $

ITEM     ident Hollerith 12 $

ITEM  heading Status 6 V(n) V(ne) V(e) V(se) V(s) V(sw) V(w) V(nw) $

ITEM     spare Boolean $
```

MODE Declarations

A MODE declaration initiates a normal mode of item description for the implicit declaration of all subsequently referenced and otherwise undefined simple (unsubscripted) items.

declaration § MODE description $

Examples

```
MODE Floating $

MODE fixed 15 Unsigned 3 Rounded $

MODE Boolean $
```

ARRAY Declarations

An ARRAY declaration describes the structure of a collection of similar values, and also provides a means of identifying this collection with a single item name. Rectangular arrays of any dimension may thus be declared. 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 elements, the value of the corresponding component of the index can only range from ∅ thru k-1.

declaration § ARRAY name$_{\text{of-array-item}}$ numbers$_{\text{of-elements-per-dimension}}$

description $

Examples

```
ARRAY       alpha  2 4 3  Floating Rounded $

ARRAY  card'image     8Ø  Hollerith 1 $

ARRAY  tic'tac'toe    3 3  Status V(empty) V(nought) V(cross) $

ARRAY   character     7 5  Boolean $
```

Named and Compound Statements

A statement must often be named to permit it to be referenced elsewhere in the program and executed out of its normal, listed sequence. Any JOVIAL statement -- simple, compound, or already named -- may be named, but a name is needed only when the statement is to be executed out of sequence.

statement § name$_{\text{of-statement}}$. statement

The statement brackets BEGIN and END allow a list of simple or compound statements, which may be interspersed with declarations, to be grouped into a single, compound statement.

statement § BEGIN [statement;declaration]s END

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.

statement § variable = formula $

Examples

```
alpha = alpha+27 $

signal = IT(M) $

weather($airbase$) = V(cloudy) $

indicator = -13 LS alpha LQ +100 LS beta LQ +198 OR legal (signal) $
```

Exchange Statements

An exchange statement exchanges the values designated by a pair of variables. The effect on either of the variables is as if each had been assigned the value designated by the other, and consequently, both variables must be the same type.

statement § variable == variable $

Examples

```
signal == card'image($27$) $

weather($airbase$) == weather($airbase+1$) $
```

IF Statements

An IF statement causes the next statement listed to be: executed if the Boolean formula it contains specifies True; or skipped if the formula specifies False.

statement § IF boolean-formula $

Example

```
compute'gross'pay. BEGIN
                   gross'pay($emp$) = hours'worked($emp$)*rate($emp$) $
                   IF hours'worked(emp$) GR 4Ø $
                        gross'pay($emp$) = gross'pay($emp$)+(hours'worked
($emp$)-4Ø)*rate($emp$)/2 $
                   END
```

GOTO Statements

A GOTO statement discontinues the execution of a set of consecutively listed statements and initiates the execution of another such set beginning at an explicitly specified statement.

statement § GOTO name$_{\text{of-statement-to-be-executed-next}}$ $

Examples

```
GOTO step7 $

GOTO compute'gross'pay $
```

FOR Statements and Subscripts

A FOR statement activates a subscript (which is an intrinsic, integer-valued, numeric variable identified by a single letter), assigns it an initial value, and causes the next (non-FOR) statement listed to be repeatedly executed one or more times. If the FOR statement contains only an initial value formula, this statement is executed once. However, if the FOR statement contains both an initial-value formula and an increment formula, the statement is repeatedly executed, and after each repetition, the subscript's value is incremented. Finally, if the FOR statement is complete and contains a limit-value formula as well, the subscript's incremented value is tested, and the loop is terminated when it exceeds the limit-value, in the positive direction if the increment was positive (GQ ϕ) and in the negative direction if the increment was negative.

A subscript's range of activity begins with the activating FOR statement and extends over any intervening FOR statements to include the first non-FOR statement listed, which then forms the loop's repeating body. A string of FOR statements may contain no more than one complete 3-formula FOR statement, and can create only a single loop, since the iteration mechanism consists of an implicit statement, automatically inserted after the repeating statement, which increments all the incrementable subscripts and then tests only the loop-controlling subscript to determine whether to repeat the loop.

variable$_{\text{of-numeric-type}}$ § letter

statement § FOR letter = numeric-formula$_{\text{initial-value}}$ ⟨, numeric-formula$_{\text{increment}}$ ⟨, numeric-formula$_{\text{limit-value}}$⟩⟩ $

Examples

```
ARRAY     alpha  199 fixed 36 Signed 15 Rounded $
ITEM       beta      fixed 36 Signed 15 Rounded $
                BEGIN
                beta = alpha($99$) $
                FOR I = 1,1,99 $
                FOR $ = 2,$+1 $
                     beta = beta+alpha($I$)/$ $
                END
```

```
ARRAY      node 25 25 Floating Rounded $
       transpose. BEGIN
                  FOR I = ∅,1,24 $
                       BEGIN
                       FOR J = I+1,1,24 $
                            node($I,J$) == node($J,I$) $
                  END  END
```

TEST Statements

The TEST statement indicates a jump from the middle of a FOR loop to the implicit subscript modify, test, and repeat statement at the loop's bottom. If a subscript letter is included, the modification of subsequently activated subscripts is omitted.

statement § TEST ⎕letter⎕ $

Examples

```
TEST $

TEST D $
```

TABLE Declarations

A table is a matrix of item-values. The rows of a table are called entries, and an entry consists of a related set of different items. Typically, entry K (item1(K), item2(K), --- , itemn(K)) would consist of values measuring the n pertinent attributes of "object" K. All the entries of a table have the same composition and structure in the sense that each consists of a similarly named and ordered set of items, declared within the BEGIN and END brackets of the TABLE declaration. The columns of a table thus consist of linear arrays of items, whose values correspond by index. A particular value of a table item is designated by item name and entry index.

declaration § TABLE [name$_{\text{of-table}}$] Variable$_{\text{length}}$;Rigid$_{\text{length}}$ number$_{\text{of-entries}}$ $ BEGIN [ITEM name$_{\text{of-table-item}}$ description $]s END

Examples

```
TABLE             Rigid 100 $
                  BEGIN
ITEM    frequency Floating Rounded $
ITEM    reporting Status V(unacceptable) V(unnecessary) V(optional)
                  V(recommended) V(compulsory) $
ITEM    operating Boolean $
                  END

TABLE   pay'roll  Variable 1000 $
                  BEGIN
ITEM    emp'name  Hollerith 18 $
ITEM    emp'code  fixed 12 Unsigned $
ITEM     pay'rate fixed 10 Unsig..ed $
ITEM     ytd'earn fixed 24 Unsigned $
                  END
```

The Modifiers NENT, ALL, and ENTRY

A vital parameter in table processing is number of entries. The functional modifier NENT allows this unsigned, integral value to be specified for rigid length tables, and designated for variable length tables (where it must be explicitly updated).

variable$_{\text{of-numeric-type}}$ § 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$_{\text{of-rigid-length-table-or-table-item}}$)

A very common loop in JOVIAL programming cycles through an entire table. Such a loop may be created with a shortened FOR statement that uses the

functional modifier ALL.

statement § FOR letter = ALL (name$_{\text{of-table-or-table-item}}$) $

As mentioned before, a table entry is a conglomeration of related items. The functional modifier ENTRY allows an entry to be considered a single value, represented by a single, composite symbol. An entry's value may be denoted by $\emptyset$ if all its items have values represented by zero; otherwise, its value is not denotable.

entry-variable § ENTRY (name$_{\text{of-table-or-table-item}}$ index$_{\text{of-entry}}$)

boolean-formula § entry-variable EQ;NQ ∅;entry-variable

statement § entry-variable = ∅;entry-variable $

statement § entry-variable == entry-variable $

Example

```
          ''Eliminate empty pay'roll entries.''
            FOR I = ALL (pay'roll) $
                 BEGIN
seek'empty.      IF ENTRY (pay'roll($I$)) EQ ∅ $
                      BEGIN
                      NENT(pay'roll) = NENT(pay'roll)-1 $
                      IF I LS NENT(pay'roll) $
                          BEGIN
                          ENTRY (pay'roll($NENT(pay'roll)$)) == ENTRY
(pay'roll($I$)) $         GOTO seek'empty $
                 END END END
```

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, by suffixing the pattern table's name with a distinguishing letter or numeral. The composition and structure of the like table's entry is then taken as being generated by the declarations describing the pattern table's entry, with the exception that item names are suffixed by the distinguishing letter or numeral. The like table may either declare its own length specifications or retain those of the pattern table.

declaration § TABLE $name_{of\text{-}pattern\text{-}table}$:letter;numeral (Variable$_{length}$; Rigid$_{length}$ $number_{of\text{-}entries}$) Like $

Example

TABLE pay'roll1 Rigid 1 Like $

OVERLAY Declarations

An OVERLAY declaration serves to arrange previously declared items, tables, and arrays in memory by allocating blocks of storage space to them. Beginning at a common origin, the data elements named in each overlay list are allocated consecutive blocks of storage, thus "overlaying" the storage allocated to the elements of other such lists in the declaration. Initial elements, whose names are preceded by either the OVERLAY declarator or the = separator, are therefore allocated storage beginning at the origin, while remaining elements, whose names are preceded by the , separator, are allocated storage after the block just assigned. An OVERLAY declaration may appear among the ITEM declarations within a TABLE declaration, arranging storage allocation for the table items in an entry.

declaration § OVERLAY [$names'_{of\text{-}items\text{-}arrays\text{-}tables}$]$^{=}$ $

Example

OVERLAY card'image,ident = pay'roll $

Initial Value Declarations

It is often necessary to declare items with specific initial values, for use as: parameters that are changed from run to run; as arrays and tables of constants; or as initial data. The initial value of a simple item may be denoted within the item declaration by a single constant, which must denote a value assignable to the item. This constant, preceded by the descriptor Preset, is usually inserted after the item description, but may replace it entirely for numeric and literal values. Table and array items, on the other hand, are initialized by constant arrays, which must be listed immediately after the table-item or array declaration, and which must correspond both in dimension and assignability to the item they preset.

declaration § ITEM name$_{\text{of-simple-item}}$ (description Preset) constant $

constant-array § BEGIN constants;constant-arrays END

Examples

```
ITEM error 1.235E-4 $

ITEM gamma Floating Rounded Preset 0. $

ITEM ident 6H(ABACUS) $

ITEM indic Boolean Preset 0 $

ARRAY beta 5 Floating $ BEGIN .86859 .28933 .17752 .09437 .19133 END

ARRAY character 7 5 Boolean $ 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
                                    BEGIN 1 0 0 0 1 END END
```

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 JOVIAL symbol. This allows the programmer to abbreviate lengthy expressions, to make simple additions to the language, and to create symbolic parameters.

declaration § DEFINE name ''signs$_{\text{except-the-symbol-}''}$:'' $

Examples

```
DEFINE card'sequence ''$tatus 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 ''128'' $
```

Closed Statements

A closed statement is, in effect, removed from the normal, listed sequence of statement executions, and may be correctly invoked only by a GOTO statement. Its normal successor is the statement listed after the invoking GOTO statement.

statement § CLOSE name$_{\text{of-statement}}$ $ statement

Example

```
CLOSE maintain'summaries $ BEGIN
                          FOR $ = ALL (frequency) $
                               BEGIN
                               IF NOT operating($$$) $
                                    ENTRY (frequency($$$)) = 0 $
                          END END
```

RETURN Statements

The RETURN statement indicates a jump from the middle to the implicit exit routine that is automatically inserted after the last listed statement of a closed statement or a procedure.

statement § RETURN $

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 a statement name is given in the STOP statement.

statement § STOP (name$_{\text{of-next-statement}}$) $

Examples

```
STOP $

STOP task'4 $
```

Alternative Statements

An alternative statement selects for execution from a set of statements that statement associated with the first True Boolean formula in a corresponding set of Boolean formulas. The effect of an alternative statement is therefore equivalent to that of the selected statement by itself.

statement § IFEITHer [boolean-formula $ statement]s ORIF END

Examples

```
IFEITHer alpha GR ∅ $ A = alpha*2 $ ORIF alpha LS ∅ $ A = alpha/2 $ ORIF
alpha EQ ∅ $ A = A+1 $ END

IFEITHer I EQ J $ diag($I,J$) = 1 $ ORIF 1 $ diag($I,J$) = ∅ $ END
```

DIRECT-Code Statements

A DIRECT statement allows the programmer to include, among the statements of a JOVIAL program, a routine coded in a "direct" or machine-oriented programming language. The effect of a DIRECT statement, being machine dependent, is undefined.

statement § DIRECT signs JOVIAL

Example

```
DIRECT   CAL    alpha       A 5-decimal digit Hollerith coded number
         XCA
         PXD
         CAQ    decbin,,5
         SLW    beta        An 18-bit binary integer
JOVIAL
```

SWITCHES

A switch computes a statement name, and a sequential formula specifies a statement name -- either directly, by name, or indirectly, by invoking a switch.

sequential-formula § $\text{name}_{\text{of-statement}}$;[$\text{name}_{\text{of-switch}}$(index)]

statement § GOTO $\text{sequential-formula}_{\text{specifying-next-statement}}$ $

Two kinds of switches may be declared: indexed switches; and item switches. The statement names computed by an indexed switch are declared as a list of sequential formulas -- which may themselves invoke switches. An n-position list (any position of which may be empty, effectively specifying the name of the statement after the switch invoking GOTO statement) is indexed (from ϕ thru n-1) by a 1-component index subscripting the switch name. The statement names computed by an item switch, however, are declared as a list of constant/sequential-formula pairs. The item name given in the switch declaration and the index (if any) subscripting the switch name together designate an item value. This value selects that sequential formula paired with a constant denoting an equal value. If no such constant was declared, then the switch effectively specifies the name of the statement after the switch invoking GOTO statement.

declaration § SWITCH name$_{\text{of-indexed-switch}}$ = (⌈sequential-formula⌋s') $

declaration § SWITCH name$_{\text{of-item-switch}}$ (name$_{\text{of-item}}$) = (⌈constant = sequential-formula⌋s') $

Examples

```
GOTO step($K-1$) $

SWITCH step = (step1,step2,step3, ,step5,step6) $

SWITCH branch (weather) = (V(stormy)=maintain summaries,V(cloudy)=step
($station-1$),V(sunny)=step4) $
```

PROCEDURES

A procedure statement invokes a procedure, which is a self-contained process with a fixed and ordered set of formal parameters, permanently defined by a procedure declaration. The calling parameters of the procedure statement are either: values, as specified by input formulas to the left of the = separator, and as designated by output variables to the right; or arrays, tables, and statements, as indicated by name. The formal parameters of the procedure declaration correspond to the calling parameters of the statement: those corresponding to values must be declared as items within the procedure heading; and those corresponding to arrays or tables must be declared as

arrays or tables to provide the procedure with a fixed definition of their structure, although no storage space is allocated for them. An output parameter which is, or which corresponds to a statement name must be suffixed by a period. The procedure itself is executed as though its formal parameters either designated calling parameter values, or were replaced with calling parameter names. To effect this, formal parameter input items are assigned corresponding calling parameter input values prior to the execution of the procedure, and formal parameter output item values are assigned to corresponding calling parameter output variables after the execution of the procedure (i.e., after the last statement listed within the procedure, after a RETURN statement, or after a GOTO statement referencing an output parameter statement name.

Identifiers declared inside a procedure, both formal parameters and otherwise, are defined for the procedure only. They bear no relation to identical identifiers declared outside the procedure, though outside identifiers may also be referenced inside.

For a procedure to specify a function value, the procedure name itself must be considered the sole, formal output parameter, and must be declared as an item in the procedure heading.

statement § name$_{\text{of-procedure}}$ ([[formula;name]s'] [= [variable;[name[.]]]s']) $

declaration § PROCedure name$_{\text{of-procedure}}$ ([names'] [= [name [.]]s']) $

[declarations] statement

Examples

```
PROCedure   factorial  (number) $
ITEM        factorial  fixed 41 Unsigned $
ITEM         number    fixed Ø4 Unsigned $
                       BEGIN
                       factorial = 1 $
                       FOR N = 1,1,number $
                            factorial = N*factorial  $
                       END
```

```
PROCedure   word'sort (word,length) $ ''A procedure that sorts a given length
                      list of 5-character words into alphabetic order, using
                      the shuttle exchange method.''
ARRAY             word ∯ Hollerith 5 $
ITEM            length fixed 15 Unsigned $
                      BEGIN
                      FOR I = ∯,1,length-2 $
                           BEGIN
                           IF word($I+1$) LS word($I$) $
                                BEGIN
                                word($I+1$) == word($I$) $
                                FOR J = I,-1 $
                                     BEGIN
                                     IF J EQ ∯ OR word($J$) GQ word($J-1$) $
                                          TEST I $
                                     word($J$) == word($J-1$) $
                      END END END END
```

INPUT/OUTPUT AND FILES

Many data storage devices impose accessing restrictions in that inserting or obtaining a particular value may 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 a string of records, which are themselves strings -- of bits or of 6-bit, Hollerith-coded bytes. (In general, records are composite symbols, which may represent entire groups of values when stored within the computer's internal memory.) A file is activated by the execution of an OPEN INPUT or OPEN OUTPUT statement, and deactivated by a SHUT INPUT or SHUT OUTPUT statement. Active files may be both written and read, one record per transfer, although some files are read-only or write-only depending on the storage device involved. An INPUT statement initiates a read operation, which transfers a record from the file into memory so as to represent a designated value or group of values, and an OUTPUT statement initiates a write operation, which transfers the record representing a specified value or group of values from memory out to the file. Such transfers are successfully terminated when the record-string is exhausted or, for a read, when it has represented all the designated values.

value-group § variable;name$_{\text{of-array-item}}$;[name$_{\text{of-table}}$ (($ numeric-formula$_{\text{indexing-first-entry}}$ [... numeric-formula$_{\text{indexing-last-entry}}$] $))]

statement § OPEN;SHUT INPUT;OUTPUT name$_{\text{of-file}}$ $

statement § INPUT name$_{\text{of-file}}$ value-group $

statement § OUTPUT name$_{\text{of-file}}$ constant;value-group $

The records of an n-record file are indexed from ∅ thru n-1, and the index of the record currently available for transfer to or from the file is designated with the file-position functional modifier, POSition. File position ranges from ∅ (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 variable that may be altered by the assignment of an arbitrary 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.

variable$_{\text{of-numeric-type}}$ § POSition (name$_{\text{of-file}}$)

Files are declared either binary or Hollerith in type, and associated with each file is a set of status constants denoting the possible states of the storage device containing the file. File status may thus be determined with a relational Boolean formula wherein the file name is considered as a status variable that is automatically updated prior to comparison according to the current state of the file's storage device. Although some of the simpler files are standard, the names of storage devices and the number and meaning of their associated states are generally computer dependent.

relational-formula § name$_{\text{of-file}}$ EQ;GR;GQ;LQ;LS;NQ status-constant

declaration § FILE name$_{\text{of-file}}$ Binary;Hollerith number$_{\text{of-records-maximum}}$ Variable$_{\text{record-length}}$;Rigid$_{\text{record-length}}$ number$_{\text{of-bits-or-bytes-per-record-maximum}}$ status-constants name$_{\text{of-storage-device}}$ $

Example

```
''This statement creates and maintains a library of up to 32768 72-character input messages,
each containing a 6-letter identifying key.  The 'key' function extracts the key word from a message
and the 'hash' function determines a 15-bit integer from a 6-letter word.''
                    BEGIN
FILE          message Hollerith 500 Rigid 72 V(inactive) V(ready) V(busy)
                    V(error) CARD'READER $
FILE          library Hollerith 32768 Rigid 72 V(inactive) V(ready) V(busy)
                    V(error) DRUM $
ITEM             card Hollerith 72 $
ITEM             line Hollerith 72 $
ITEM             size ''of library'' fixed 16 Unsigned $
                    OPEN INPUT message $  OPEN OUTPUT library $
                    IF size LS 32768 $
                        BEGIN
                step3.  INPUT message card $
                step4.  IF message EQ V(ready) $
                            BEGIN
                            size = size+1 $
                            FOR I = hash (key (card)) $
                                BEGIN
                step7.                  POSition (library) = I $
                step8.                  INPUT library line $
                step9.                  IF library EQ V(ready) $
                                            BEGIN
                                            IF key (line) EQ 6H(          ) OR key
(line) EQ key (card) OR hash (key (line)) NQ I $
                                                BEGIN
                                                POSition (library) = I $
                                                OUTPUT library card $
                                                IF key (line) EQ key (card) $
                                                    BEGIN
                                                    size = size-1 $
                                                    GOTO step3 $
                                                    END
                                                IF key (line) EQ 6H(          ) $
                                                    GOTO step3 $
                                                card = line $
                                                END
                                            GOTO step8 $
                                            END
                                        IF library EQ V(busy) $  GOTO step9 $
                                        IF POSition (library) EQ 32768 $
```

Page 36 blank

```
                    BEGIN
                    i = # $ GOTO step7 $
               END END END
          IF message EQ V(busy) GOTO step4 $
          IF message EQ V(error) OR library EQ V(error) $
               GOTO correction routine $
          END
     SHUT INPUT message $ SHUT OUTPUT library $
     END
```

PROGRAMS

A JOVIAL program is a list of declarations and statements enclosed in the brackets START and TERM. 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. The termination separator $ indicates the typographic end of the program.

program § START declarations statements TERM [name$_{\text{of-first-statement}}$]$

INDEX OF SYMBOLS AND TERMS

The following index gives page numbers for the grammatical rules containing the JOVIAL symbols and terms used in this document.

Defined on Page	Symbol or Term	Used on Page
	GOTO	7,20,29
	GQ	7,33
	GR	7,33
	Hollerith	7,10,33
	IF	7,20
	IFEITHer	7,28
11	index	11,12,24,29
	INPUT	7,33
8	integer	9,14
	ITEM	7,17,23,26
	JOVIAL	7,29
6	letter	7...9,21,22,24,25
	Like	7,25
9	literal-constant	9,15
15	literal-formula	16
7	logical-operator	
	LQ	7,33
	LS	7,33
	MANT	7,12
	MODE	7,17
8	name	9,11...13,17...20,23...31,33,35
	NENT	7,23
	NOT	7,16
	NQ	7,24,33
8	number	8...10,18,23,25,33
6	numeral	7,8,25
14,23	numeric-formula	11,12,14,16,21,33
9	octal-constant	9,14,15
	OPEN	7,33
	OR	7,16
	ORIF	7,28
	OUTPUT	7,33
	POSition	7,33
	Preset	7,26
	PROCedure	7,31
35	program	
16,33	relational-formula	16
7	relational-operator	16
	RETURN	7,28
	Rigid	7,23,25,33
	Rounded	7,10
	SHUT	7,33
7	sign	9,10,27,29

Defined on Page	Symbol or Term	Used on Page
	Signed	7,10
7	separator	
29	sequential-formula	29,30
7	sequential-operator	
	START	7,35
19...22,24,28,29,31,33	statement	19,28,31,35
	Status	7,11
9	status-constant	9,10,15,33
15	status-formula	16
	STOP	7,28
	SWITCH	7,30
	TABLE	7,23,25
	TERM	7,35
	TEST	7,22
	Transmission	7,10
	Unsigned	7,10
33	value-group	33
	Variable	7,23,25,33
11,12,21,23,33	variable	14...16,19,31,33
	)	7,12...14,16,23,24,
	-	7...10,14
	+	7...10,14
	=	7,19,21,24,30,31
	==	7,19,24
	$	7,10,17,...31,33,35
	$)	7,11,12,33
	*	7,14
	**	7,14
	(	7,12...14,16,23,24,3
	($	7,11,12,33
	(/	7,14
	,	7,11...13,21,30,31
	:	7,8
	' '	7,10,27
	/	7,14
	/)	7,14
	.	7,9,19,31
	...	7,33