TM-555/002/01

The JOVIAL Manual: Part 2, Revision 1

The JOVIAL Grammar and Lexicon

9 June 1961

TM-555/002/01

TECHNICAL MEMORANDUM

(TM Series)

THE JOVIAL MANUAL:

Part 2, Revision 1

THE JOVIAL GRAMMAR AND LEXICON

C. J. Shaw

9 June 1961

SYSTEM
DEVELOPMENT
CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA

Preface

Part 2, The JOVIAL Grammar and Lexicon, is a complete, concise, and rigorously formal description of JOVIAL, an SDC-designed, procedure-oriented, programming language. It is intended mainly as a reference for programmers already familiar with the language.

The Grammar describes the syntax or appearance of JOVIAL, while the Lexicon describes the semantics or meaning of JOVIAL. The Lexicon, which is interleaved with the Grammar, attempts to ascribe a precise meaning to every 'grammatical' JOVIAL program by describing the computing rule denoted by the program. In this it is not entirely successful, for the interpretation of some few JOVIAL expressions (especially in the input-output area) necessarily varies from computer to computer so that these expressions must remain undefined. Machine independent programming can be achieved, however, by avoiding undefined constructions.

Any grammatical construction not explicitly defined in the Lexicon, as well as every ungrammatical construction, is to be considered undefined, which merely means that its effect on the operation of a program containing it is computer dependent. All ungrammatical constructions and many undefined constructions are mistaken with regard to the intent of the programmer, and may even be illegal -- flagged as mistakes by the compiler.

Table of Contents

Part 2. THE JOVIAL GRAMMAR AND LEXICON

A Complete, Concise, & Formal
Technical Description

Languages have three different aspects, representing successive levels of abstraction: pragmatics, the language's use--as described in a Primer; semantics, the language's meaning--as described in a Lexicon; and syntax, the language's appearance--as described in a Grammar. Languages must be described in language, and it is convenient to call the language being described the 'object-language', and the describing language, the 'metalanguage'. In the present case, the object-language is, of course, JOVIAL. The metalanguage of the JOVIAL Primer is English and JOVIAL, the metalanguage of the JOVIAL Lexicon is English, and the metalanguage of the JOVIAL Grammar is a specially developed syntax metalanguage.

The Syntax Metalanguage

A language consists of certain signs, and certain strings of these signs. An object-language sign-string may be exhibited, by writing its signs in linear order, or it may be denoted, by a metalanguage sign-string. In describing the syntax metalanguage, both exhibited and denoted object-language sign-strings will be called symbols and may be distinguished by the fact that the object-language signs and the metalanguage signs do not intersect. A symbol, or a sequence of symbols delimited by the metalinguistic connectors ":" or ";", is a form, and the following grammar is just a set of rules for substituting forms for metalanguage symbols.

Notation

:	is the catenation operator, denoting the juxtaposition of symbols. Thus, the form "A:B", which may be read "A concatenated with B", denotes "AB".

; is the selection operator, denoting a choice of symbols. Thus, the form "A;B", which may be read "A or B", denotes either "A" or "B". Selection has precedence over catenation, so that the form "A:B;C:D", which may be read "A, concatenated with B or C, concatenated with D", denotes either "ABD" or "ACD".

⸢Φ⸣ is a symbol denoting a single form, Φ. The ⸢ and ⸣ serve only as metalanguage brackets.

{Φ} is a symbol denoting a string of forms Φ of arbitrary length one or greater, i.e., "⸢Φ⸣;⸢Φ:Φ⸣;⸢Φ:Φ:Φ⸣;⸢Φ:Φ:Φ:Φ⸣; etc." (A superscript number may be used to denote a string of forms of specific length, e.g., "$\{Φ\}^7$" signifies "Φ:Φ:Φ:Φ:Φ:Φ:Φ".)

∅ is the null symbol, used mainly to construct optional forms. Thus, the form "A:∅:B" denotes "AB", and "A:∅;B" denotes either "A" or "AB".

↓ may be read "can be rewritten as".

To create a semi-pictorial grammar and thus facilitate legibility, the following pair of definitions are introduced, augmenting the metalanguage:

1. The symbol ␢ will denote a single blank character.

2. The blank will be used as a special catenator, denoting the linkage of a symbol pair by a string of blanks of arbitrary length zero or greater, or when used to concatenate a numeral-letter pair, of arbitrary length one or greater. Thus "A B", which may be read "A followed by B", denotes "A:{␢}:B", and "A 8" denotes "A:{␢}:8", while "A \$" denotes "A:∅;{␢}:\$".

A number subscripting a symbol has no syntactic significance, but serves only to distinguish identical symbols with different meanings in the semantic description. The remaining notational conventions should be self-explanatory.

To illustrate the syntax metalanguage, consider the language L, which consists of the three signs $,), and (, and the following sign-strings:

	(),	(()),	((())),	(...()...),
$,	($),	(($)),	((($))),	(...($)...),
$$,	($$),	(($$)),	((($))),	(...($$)...),
$$$,	($$$),	(($$$)),	((($$$))),	(...($$$)...),
$...$,	($...$),	(($...$)),	((($...$))),	(...($...$)...).

The syntax of L may be completely described by the grammatical rule:

$$\underline{L} \quad \Updownarrow \quad \lfloor\,\rfloor;\lfloor\$\rfloor;\lceil(:\underline{L}:)\rceil$$

which may be read "L can be rewritten as nothing, or $-string, or (concatenated with L concatenated with)".

Note: A language whose syntax can be entirely described with the above metalanguage is a 'formal' language; a formal language whose syntax can be entirely described when the metalinguistic connector denotes only the rewriting of a single metalanguage symbol is a 'phrase-structure' language; a phrase-structure language whose syntax can be entirely described in terms of exhibited sign-strings is a 'finite state' language. Both L and JOVIAL are phrase-structure languages, but not finite state languages.

General

JOVIAL is an ALGOL-type, procedure-oriented programming language designed to permit simple and concise description of certain basic information processing operations. These operations are:

1. Logical decision;
2. Specification of values for variable items of information;
3. Calculation necessary for both 1 and 2;
4. Input and Output of information.

A JOVIAL program* or procedure is a sequence of statements, which

*Terms for which a precise definition will be given later are underlined at their first occurrence.

may be supported by <u>declarations</u>. Each statement describes a rule of computation and either explicitly or implicitly specifies a successor statement, which is the next statement listed unless otherwise stated. The computing rule given by a program is thus the sequence of computations described by the statements, taken in the order supplied by successor relations. Declarations, which are not themselves computing rules, serve to describe the environment in which the computing rules are to operate by defining certain properties of the <u>identifiers</u> appearing in the statements.

THE ALPHABET

Letter.

Form ØØ1. @ ⫲ 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.

Form ØØ2. # ⫲ Ø;1;2;3;4;5;6;7;8;9

Mark.

Form ØØ3. & ⫲ ≡;);(;+;-;*;/;.;,;';=;$

Sign.

Form ØØ4. sign ⫲ @;#;&

At the most basic level, a JOVIAL program consists of a single, linear string of signs -- segmented for convenience only into lines written on a coding sheet or punched in a card. Signs do not have individual meaning, but are used for forming the symbols of JOVIAL, that is, <u>delimiters</u>, identifiers, and <u>constants</u>.

THE VOCABULARY

Signs are grouped into symbols, the words of JOVIAL. Symbols contain no embedded blanks, since they may be separated from each other by an arbitrary number of blanks. However, such separation is necessary only when both the adjacent signs being separated are numerals or letters. Where necessary, symbols may extend past the end of a line.

DELIMITERS

Form ØØ5. Arithmetic Operator ǂ +;-;*;/;**

Form ØØ6. Relational Operator ǂ EQ;GR;GQ;LQ;LS;NQ

Form ØØ7. Logical Operator ǂ AND;OR;NOT

Form ØØ8. Sequential Operator ǂ GOTO;IF;IFEITH;ORIF;FOR;TEST;
CLOSE;RETURN;STOP

Form ØØ9. File Operator ǂ INPUT;OUTPUT;OPEN;SHUT

Form Ø1Ø. Functional Modifier ǂ BIT;BYTE;CHAR;MANT;ODD;
ALL;POS;ENTRY;NENT;NWDSEN;A

Form Ø11. Separator ǂ .;,;=;==;';...;$;ASSIGN

Form Ø12. Bracket ǂ (;);(/;/)($;$);'';'';BEGIN;END;START
TERM;DIRECT;JOVIAL

Form Ø13. Declarator ǂ MODE;ITEM;TABLE;STRING;ARRAY;FILE;
SWITCH;PROC;DEFINE;OVERLAY

Form ∅14. <u>Abbreviation</u> ⧫ A;B;D;E;F;H;L;M;N;O;P;R;S;T;U;V

Delimiters have a fixed meaning, which is usually obvious, and is best described in the context of the meaning of their use in later forms.

IDENTIFIERS

Label.

Form ∅15. label ⧫ @:($_0$($_0$;':@;#)$_0$ Except delimiters

Examples

AW3W

U2

NUMBER'OF'SIGNIFICANT'CORRELATIONS

ALPHA

Item Name.

Form ∅16. itemname ⧫ fname;aname;dname;lname;bname;sname

Floating Item Name.

Form ∅17. fname ⧫ label

Fixed (Arithmetic) Item Name.

Form ∅18. aname ⧫ label

Dual Item Name.

Form ∅19. dname ⧫ label

Literal Item Name.

Form ∅2∅. lname ⧫ label

Status Item Name.

Form Ø21. sname ‡ label

Boolean Item Name.

Form Ø22. bname ‡ label

Table Name.

Form Ø23. tablename ‡ label

File Name.

Form Ø24. filename ‡ label

Statement Name.

Form Ø25. statementname ‡ label

Switch Name.

Form Ø26. switchname ‡ label

Procedure Name.

Form Ø27. procedurename ‡ label

Identifiers serve to label the various elements of the program's environment and have no inherent meaning, only that supplied them by their defining declarations (or statements, in the case of statement names.) All identifiers except statement names must be defined by an appropriate declaration, which may be from a compool or list of system declarations. Every identifier in a JOVIAL program must be unique within its <u>scope</u>, in that no other identifier denoting a different element with an overlapping scope may have the same spelling.

CONSTANTS

Constants serve to denote specific values, represented by specific machine language symbols.

Constant.

Form ∅28. constant ⩴ icon;fcon;acon;ocon;dcon;lcon;bcon

Number.

Form ∅29. n ⩴ ⦅#⦆

Examples
∅
123456789

Integer Constant.

Form ∅3∅. icon ⩴ ⦅⦆;+;-:n:⦅⦆;⁅E:n⁆

Examples
27
-∅∅39
+331E9

Floating Constant.

Form ∅31. fcon ⩴ ⁅⦅⦆;+;-:n:.:⦅⦆;n⁆;⁅⦅⦆;+;-:.:n⁆:⦅⦆;⁅E:⦅⦆;+;-:n⁆

Examples
27.
-∅.∅39
+3.31E-6

Fixed (Arithmetic) Constant.

Form ∅32. acon ⩴ fcon:A:⦅⦆;+;-:n

Examples
27.AØ
-.ØØ39A11
+3.31E6A-8

Numbers, and integer, floating, and fixed constants have a conventional, decimal meaning. The number following the abbreviation E (for Exponent) in some integer, floating, and fixed constants is a scale factor, expressed as an integral power of 1Ø multiplier. Thus, -3.31E-6 denotes the same floating-point numeric value as -.ØØØØØ331, and 3.31E6 the same as 331ØØØØ. The number following the abbreviation A in all fixed-point constants indicates the precision requirement, the number of fractional bits (bits After the binary point) in the machine language representation of the fixed-point numeric value. A negative number following the A merely indicates that the value (of the least significant bit) is precise to the corresponding positive power of two.

Octal Constant.

Form Ø33. ocon ∮ O(:{Ø;1;2;3;4;5;6;7}:)

Octal constants have the obvious meaning of octal integers.

Examples
O(Ø127)
O(7773Ø5)

Dual Constant.

Form Ø34. dcon ∮ D(:⸢icon:,:icon⸣;⸢acon:,:acon⸣;⸢ocon:,:ocon⸣:)

Dual constants denote vector or complex quantities, with the constants comprising the component pairs having the meanings already described. If the component pair of constants are fixed constants, their precision requirements (indicated number of fractional bits) must be the same.

Examples

```
D(27,-Ø39)
D(-1.384E3A-2,+6.A-2)
D(O(Ø127),O(77713))
```

Literal Constant.

Form Ø35. lcon ⇕ n:H;T:(:$\{$sign$\}_0^n$:)

Literal constants denote strings of JOVIAL signs--represented in machine language by a six bit-per-sign code. The initial number indicates the number of signs, written inside the parentheses, to be represented. Two coding schemes are used: Hollerith, indicated by the abbreviation H; and Transmission code, indicated by the abbreviation T. Hollerith code has an undefined numeric representation, while Transmission code has the numeric representation given in the following chart.

STANDARD TRANSMISSION CODE

FIRST OCTAL DIGIT	Ø	1	2	3	4	5	6	7
Ø		▨	▨	▨	▨	▨	A	B
1	C	D	E	F	G	H	I	J
2	K	L	M	N	O	P	Q	R
3	S	T	U	V	W	X	Y	Z
4	)	-	+	▨	=	▨	▨	$
5	*	(	▨	▨	▨	▨	,	▨
6	Ø	1	2	3	4	5	6	7
7	8	9	'	▨	/	.	▨	▨

SECOND OCTAL DIGIT

Examples

```
48H(THIS IS A (+-*/.,='$) HOLLERITH LITERAL CONSTANT)
45T(THIS IS A TRANSMISSION CODE LITERAL CONSTANT.)
1ØH(Ø123456789)
1T(()
```

Boolean Constant.

Form Ø36. bcon ∮ Ø;1

Boolean constants denote the logical values true and false of Boolean algebra. True is denoted by 1 and false is denoted by Ø. They are distinguished from the integers one and zero by context.

Status.

Form Ø37. status ∮ V(:@;label:)

A status is, in essence, a mnemonic label denoting one of the values of a status <u>item</u>, or one of the states of a <u>file</u>. Statuses are useful in those cases where mnemonic labels are more intelligible than numeric constants in denoting an item's value. The actual, numeric value denoted by a status is implicitly defined in the declaration of either the item or the file associated by context with the status. A status may not duplicate other statuses belonging to the same status item (or file), but may duplicate statuses belonging to other status items (or files) since statuses are defined only in connection with a specific status item (or file).

Examples

```
V(GOOD)
V(THIS'IS'A'STATUS)
V(X)
```

CLAUSES

Symbols are grouped into <u>variables</u> and <u>formulas</u>, which are <u>expressions</u>, and into <u>phrases</u>. Expressions and phrases are the clauses of JOVIAL.

COMMENT.

Form ∅38. (‡) ‡ '':(sign):'' Where (sign) excludes the symbols '' and $

A comment serves only to allow remarks or clarifying text to be included among the symbols of a program and has no operational significance or effect on the program. With the sole exception of the DEFINE declaration (form 114), a comment may appear wherever an arbitrary length string of blanks would be grammatical, that is, between symbols.

Example
''THIS IS A REMARK.''

EXPRESSIONS

Expressions are said to express values (in general, quantities denotable by constants) which may vary during the execution of the program.

Variable.

Form ∅39. variable ‡ avar;lvar;svar;bvar;entvar

A variable designates a single value, which may be used to compute other values, and which may be altered by the execution of an <u>assignment</u> statement, an <u>exchange</u> statement, a procedure statement, or an INPUT statement.

Arithmetic Variable.

An arithmetic variable designates a numerical value, of the type denotable by integers, floating constants, fixed constants, octal constants, or dual constants; that is, the class of rational numbers or rational number pairs.

Form ∅4∅. avar ǂ fname;aname;dname ℓ;ᑭ($ index $)ᑫ

Floating, fixed, and dual items are arithmetic variables. Dual items are similar, fixed-point item pairs. The terms 'floating-point' and 'fixed-point' refer to two systems of representing rational numbers in machine language. In floating-point, scaling information is carried as part of the physical representation of the value and is manipulated either by the circuitry of the computer or by subroutines. In fixed-point, scaling information must be implicit in the instruction logic of the machine-language program.

Where the item name is subscripted, the <u>index</u>, enclosed in the subscription brackets ($ and $), indicates a single value from the set of values belonging to a <u>table</u> item, a <u>string</u> item, or an <u>array</u> item.

Examples

```
ALPHA
GROSS'PAY($E$)
R273($N+2, PRCA($H$)**3$)
```

Form ∅41. avar ǂ @

Subscripts are identified by single letters and are integer valued arithmetic variables designating both positive and negative integers. Truncation of the fractional portions of any non-integer value assigned to a subscript is implied. Subscripts, so-called because their most typical use is specifying index values within subscriptions, are defined by <u>FOR</u> statements, and their scope of definition includes just the next statement listed, and the scope of definition of any subscript defined in that statement.

Examples

I

J

Form ∅42. avar ⸸ BIT ($ index $) (itemname ∂;⁅($ index $)⁆)

The functional modifier BIT, with its associated subscription, operating on any simple or subscripted item is an arithmetic variable designating a positive, integral value. The machine language representation of the value of any item is a string of bits, indexed from left (most significant) to right (least significant). For an n-bit item, the index values range from ∅ thru n-1. The 1 or 2-component index subscripting the BIT modifier indicates a substring of the bits representing the value of the item, taken as an unsigned integer. The first value of the index indicates the initial bit of the substring. For a 2-component index, the second value indicates the number of bits in the substring. For a 1-component index, where the second value is omitted, the length of the substring is implicitly 1, so that a 1-bit integral value is designated.

The value of a BIT variable is defined only if the index subscripting the BIT modifier indicates a substring of bits within the bit range of the item. Furthermore, although the value of a bit-string of length zero is zero, the value of a bit-string of negative length is undefined. The value of a BIT variable over a signed arithmetic item is also undefined, being computer dependent because of the varying schemes for representing negative numbers.

Examples

```
BIT($9$)(ALPHA)
BIT($ALPHA/2,I*7$)(GAMMA6∅($BETA($I,K$)+3$))
BIT($BIT($∅$)(DELTA), N'DELTA-BIT($∅$)(DELTA)$)(DELTA)
```

Form ∅43.   avar ‡ CHAR;MANT (fname ⸢;⸢($ index $)⸣)

The machine language representation of the value of a floating-point item consists of two essentially fixed-point components: the mantissa, a signed fraction; and the characteristic, a signed integer scaling factor or power of two multiplier for the mantissa. The functional modifier CHAR or MANT, operating on a simple or subscripted floating-point item, is an arithmetic variable designating the signed integral value of the characteristic in the case of CHAR, or the signed fractional value of the mantissa in the case of MANT.

Examples

```
CHAR(ALPHA)
MANT(R274($I,J,K$))
CHAR(BETA($MANT(ALPHA)$))
```

Form ∅44.   avar ‡ POS (filename)

A file, consisting of a sequence of <u>records</u>, is a self-indexing storage device, in the sense that the record available for transfer to or from the file depends on the current position of the file. The File Position functional modifier POS, operating on a file name, is an arithmetic variable designating a positive, integral value which determines or is determined by the current position of the file. For a file consisting of k records, this value ranges from ∅ thru k, corresponding to the k+1 positions of the file. File position is defined only for <u>active</u> files.

Example

```
POS(PERSONNEL'FILE)
```

Form ∅45.   avar ‡ NENT (tablename;itemname)

The functional modifier NENT (<u>N</u>umber of <u>ENT</u>ries), operating on a table name or a table item name, is an arithmetic variable designating a positive, integral value: the number of entries in the indicated table. Actually, NENT acts as a true variable only if the indicated table is a variable length table, or if the indicated table item is an item in a variable length table. In all other defined cases, NENT must be considered an unsigned integer constant.

Example
NENT(DATA8)

Literal Variable.

A literal variable designates as its value a string of JOVIAL signs. The value of a literal variable is denotable by either a literal constant or an octal constant.

Form ∅46. lvar ⇞ lname ⌊;⌈($ index $)⌉

Literal items, both simple and subscripted, are literal variables designating as values strings of JOVIAL signs.

Examples
WORD
EMPLOYEE'NAME(E)
A273(N,M)

Form ∅47. lvar ⇞ BYTE ($ index $) (lname ⌊;⌈($ index $)⌉)

The functional modifier BYTE, with its associated subscription, operating on any simple or subscripted literal item, is a literal variable designating a substring of the item modified. The BYTE modifier functions in a manner entirely analogous to the operation of the BIT modifier. The machine language representation of a literal item is a string of bytes -- each byte itself a bit-string of length 6 representing a single sign. The bytes of an n-byte literal item are indexed from left to right from ∅ thru n-1. The 1- or 2-component index subscripting the BYTE modifier indicates a sub-string of the bytes representing the value of the item modified.

The first value of the index indicates the initial byte of the substring. For a 2-component index, the second value indicates the number of bytes in the substring. For a 1-component index, where the second value is omitted, the length of the substring is implicitly 1, so that a 1-byte literal value is designated.

The value of a BYTE variable is defined only if the index subscripting the BYTE modifier indicates a substring of bytes within the byte range of the item. Furthermore, the value of a byte-string of length zero is blanks, and the value of a byte-string of negative length is undefined.

Examples

```
BYTE($Ø$)(WORD)
BYTE($DELTA+K/2,4$)(EMPLOYEE'NAME($E-1$))
BYTE($N,M$)(A273($N,M$))
```

Status Variable.

A status variable designates a 'symbolic' value, essentially a mnemonic label. The value of a status variable is denotable by a status.

Form Ø48. svar ⟠ sname ∮;⌜($ index $)⌝

Status items, both simple and subscripted, are status variables.

Examples

```
WEATHER
DEPARTMENT($E$)
A275($I+1, J, J-1$)
```

Boolean Variable.

A Boolean variable designates one of the logical values, true or false, of Boolean algebra. The value of a Boolean variable can be denoted by a Boolean constant: true by 1; and false by ∅.

Form ∅49. bvar ⧫ bname ℓ;ᑭ($ index $)ᑫ

Boolean items, both simple and subscripted, are Boolean variables.

Example
BOOL
EXEMPT(E)

Form ∅5∅. bvar ⧫ ODD (@;ᑭfname;aname ℓ;ᑭ($ index $)ᑫᑫ)

The functional modifier ODD, operating on a subscript or a floating or fixed item, is a Boolean variable. ODD designates the value true when the least significant bit of the modified subscript or item represents a magnitude of one, and false when it represents a magnitude of zero. ODD is true, therefore, when the magnitude of the modified value is odd, taken as an integer, and false when that magnitude is even.

Examples
ODD(K)
ODD(CHI(K))

Entry Variable.

A table is an ordered set of entries (indexed from ∅ thru n-1 for an n-entry table) and an entry is again an ordered set--of related items. An entry variable designates as its value a conglomeration of the values comprising an entry in a table. The value of an entry variable depends both on the structure of the entry and on the values of the constituent items, and is denotable by ∅ (meaning that all items in the entry have values represented by zero) or is not denotable at all in JOVIAL.

Form ∅51. entvar ǂ ENTRY (tablename;itemname ($ index $))

The functional modifier ENTRY, operating on a subscripted table name or table item name, is an entry variable. The table name or table item name serves to indicate the table, while the 1-component index of the subscription indicates the particular entry.

Example

```
ENTRY(BETA($(/DELTA-C**2/)$))
```

Formula.

Form ∅52. formula ǂ aform;lform;sform;bform

A formula specifies a single value, either by simply designating it, or by describing the process of computing it. A formula may contain variables, and the value it specifies is, as will be described, generally dependent on the values designated by those variables.

Function.

Form ∅53. function ǂ procedurename ($\{_0$;{formula;itemname;tablename; statementname $\{_0$;{ , formula;itemname;tablename;statementname$\}_0$})

A function specifies the value arising from the application of the algorithms defining the procedure to the <u>calling parameters</u> within the parentheses -- the labels and the values specified by the formulas. To be more explicit, a function actually refers to an item, with the same name as the procedure itself, declared within the procedure. The value specified by the function is thus the value designated by this item following the execution of the procedure. The formula type in which a function appears must therefore be compatible with its synonymous item.

Examples

```
RANDOM()
SEQUEL(E)
PREDICATE(K-BETA($DELTA$)**.5A1, BYTE($L$)(WORD), V(SOON), AA($BAKER$)
NQ 37.3 OR BOOL, AIRBASE'CENTRAL, SORT)
```

Arithmetic Formula.

An arithmetic formula specifies a numerical value, of the type designated by arithmetic variables.

Form ∅54. aform ⧲ icon;fcon;acon;ocon;dcon;avar;function

When the arithmetic formula consists of just a single operand, the value it specifies is obvious: that value denoted by the constant, or designated by the variable, or specified by the function.

Examples

```
-1.357E4A7
W
EMPLOYEE'NUMBER($E$)
BIT($∅,N'BETA$)(BETA($1,ALPHA/W,W$))
SIN(OMEGA*T**2)
```

Form ∅55. aform ⧲ NWDSEN (tablename;itemname)

The functional modifier NWDSEN (Number of WorDS per ENtry), operating on a table name or a table item name, is an arithmetic formula specifying a positive, integral value equal to the number of storage units, called words or registers, allocated per entry of the indicated table.

Example

```
NWDSEN(ABLE'TABLE)
```

Form ∅56. aform ⧲ ⸢+;- aform⸣;⸢(aform)⸣;⸢(/ aform /)⸣;
⸢aform +;-;*;/;** aform⸣

An arithmetic formula containing one or more arithmetic operators specifies the value arising from the computation(s) described by the formula, in the familiar sense as defined by the notation of ordinary algebra. The binary arithmetic operators

+, -, *, /, and ** have the conventional meanings addition, subtraction, multiplication, division, and exponentiation. (Because of the limited range of the JOVIAL sign set, the symbols * for multiplication--which for syntactic reasons must always be explicitly denoted--and ** for exponentiation necessarily differ from conventional notation.) The unary arithmetic operator - denotes the operation of negation, and the unary operator +, though redundant, is allowed. The parentheses perform their usual grouping function, and the bracket pair (/ and /) denote the absolute magnitude of the value specified by the arithmetic formula they enclose. Division by zero is undefined, as is an exponentiation such as (-2)**.5 which would result in a complex root being taken.

Scope. The scope of any operation in an arithmetic formula may be indicated by the use of parentheses (or absolute magnitude brackets.) In the absence of bracketing, negations are performed first, then exponentiations, then multiplications and divisions, and finally, additions and subtractions. Within these categories, the sequence of operations is from left to right. These rules of precedence allow any desired sequence of operations, provide an unambiguous indication of the scope of each operation, and conform to customary usage.

Mode. Arithmetic computations are performed in one of three modes of arithmetic: floating-point; fixed-point; or dual fixed-point. (Dual mode arithmetic is performed on dual, fixed-point operands, the indicated operation being performed individually on both the left and right components of the operands involved.) Mode selection for a particular operation is based on the operands involved and on the intended use of the result. In general: operations involving only floating operands will be performed in the floating-point mode; operations involving only fixed operands will be performed in the fixed-point mode; and operations involving dual operands will be performed in the dual mode. Operations involving both floating and fixed operands will be performed in a mode compatible with the intended use of the result, with an automatic conversion between floating and fixed representation implied. Operations involving a dual operand will be performed in the dual mode and will yield a dual result, with any mono-valued operands involved being 'twinned' in the process to yield a dual value.

Precision. Floating-point computations are carried out with numbers of fixed though undefined significance, and varying precision. Fixed-point computations, on the other hand, involve numbers of fixed precision, and varying significance. Since the size of the result of a fixed-point arithmetic operation may not exceed the least multiple of the computer's word size that can contain the most significant operand, results with greater possible significance must be truncated to this limit. This truncation is performed first on the least significant fraction bits and then, if necessary, on the most significant integer bits. In determining the maximum possible significance of a fixed-point result, integers are regarded as arbitrarily precise.

Examples

```
ALPHA - SIN(OMEGA * T ** 3) ** 2
W ** (SIN(ZYME5($A$)) + 2.93)
(-ALPHA * 9 + (/CHI($F+6$)/) ** BIT($Ø,K$)(RB276($I,J-1,K$)))/2
```

Index.

Form Ø57. index $\doteq$ aform $\}_0$;$(_0$, aform$)_0$

An index, composed of an ordered list of arithmetic formulas separated by commas, is a vector whose components are positive (non-negative) integral values. (Consequently, any fractional part of the value specified by an arithmetic formula in an index list is truncated.) An index serves to indicate a particular value in the designation of values which are elements of tables, of strings or of multi-dimensional arrays. Such a value is designated by an index subscripting an item name. Thus, a simple item is unsubscripted; a table item is subscripted by a 1-component index indicating the table entry in which the value occurs; a string item is subscripted by a 2-component index, the first value indicating the 'bead' or element within the string and the second value indicating the table entry in which the value occurs; and a multi-dimensional array item is subscripted by a correspondingly multi-component index, the first value indicating the row, the second value indicating the column, the third value indicating the plane, and so on.

An index, when subscripting a BIT or BYTE functional modifier, also serves to indicate a substring of the bits or bytes representing the value of the item being modified, the first value indicating the initial bit or byte, and the second value indicating the number of bits or bytes -- with an omitted second value implying a substring or length 1.

An index is undefined if the number of components does not correspond to the dimensionality of the item (or functional modifier) it subscripts, or if any of its component values fall outside the range bounded by the index limits (ø and n-1 for an n-element set) of the corresponding dimension of the item (or functional modifier) being subscripted.

Examples

```
ø, 1, 2
BETA($J$)
K,J,I-3*DELTA,(/R273($I,2*J,K/2$)/),ø
```

Literal Formula.

Form ø58. lform ∮ lcon;ocon;lvar;function

The value specified by a literal formula is that value denoted by the literal or octal constant, or designated by the literal variable, or specified by the literal function.

Examples

```
17H(THIS IS AN LFORM.)
11T(SO IS THIS.)
O(77)
WORD($ALPHA$)
BYTE($K**.5A1$)(BETA($A,A-I/2+DELTA$))
TAIL(BYTE($A,B$)(SYMBOL), O(174), 18)
```

Status Formula.

Form ∅59.   sform ‡ status;svar;function

The value specified by a status formula is that value denoted by the status, or designated by the status variable, or specified by the status function.

Examples

```
V(AMAZING)
WEATHER($K$)
TYPE(WORD($COUNT$))
```

Boolean Formula.

The value specified by a Boolean formula is one of the values true (1) or false (∅) of Boolean algebra.

Form ∅6∅.   bform ‡ bcon;bvar;function

When the Boolean formula consists of just a single Boolean operand, the value it specifies is that value, true or false, denoted by the Boolean constant, or designated by the Boolean variable, or specified by the Boolean function.

Examples

```
1
BOOL
INDICATOR($I,J,K**3$)
SIGN(ALPHA)
ATOM(CDR(FORM($K$)))
EQUAL(BYTE($I+1$)(ALPHA), BYTE($GAMMA$)(ALPHA))
```

Form Ø61. bform ‡ aform $\{_0$ EQ;GR;GQ;LQ;LS;NQ aform $\}_0$

A relational operator denotes a relation between the values specified by the pair of formulas immediately bracketing it. A simple Boolean formula (i.e., with no logical operators) containing relational operators specifies the value true whenever all the corresponding relations are satisfied for the formulas involved, otherwise, the value specified is false.

The relational operators denote primarily numeric relations:

EQ	is EQual to
GR	is GReater than
GQ	is Greater than or eQual to
LQ	is Less than or eQual to
LS	is LesS than
NQ	is Not eQual to

Between arithmetic values, the meaning of these relations is fairly obvious: the test of a relation being implemented by an arithmetic comparison. Between dual-valued arithmetic formulas, the relation, to specify the value true, must hold for both component pairs, and between a single-valued and a dual-valued formula, it must hold between the single value and both halves of the dual value. The precision of an arithmetic comparison, though undefined, is at least as precise as the least precise of the two values involved.

Examples

```
X EQ -2
(/BETA($I$) - BETA($J$)/) LQ I + J
ALPHA GR NENT(BETA) LQ R273($I,I$)+49 LS Ø
```

Form Ø62. bform ∮ lform $(_{0}$ EQ;GR;GQ;LQ:LS:NQ lform$)_{0}$

Examples

```
ALPHA EQ BETA($H$)
ALPHA GR 5T(GREEN)
BYTE($QR4($K$)$)(ALPHA) NQ O(77)
BYTE($Ø,3$)(ALPHA) NQ BYTE($J,3$)(BETA($H$)) NQ 3H(...)
```

Form Ø63. bform ∮ svar;filename EQ;GR;GQ;LQ;LS;NQ sform

Examples

```
WEATHER EQ V(FAIR'AND'WARMER)
BETA($2*DELTA,Ø$) GR V(A34)
PERSONNEL'FILE NQ V(ACTIVE)
```

Form Ø64. bform ∮ entvar EQ;NQ entform

Examples

```
ENTRY(BETA($K$)) EQ Ø
ENTRY(DELTA($I$)) NQ ENTRY(DELTA($Ø$))
```

Between non-arithmetic values of the literal, status, or entry type, the meaning of the relations denoted by the relational operators is based on the machine language representation of the values involved. For purposes of comparison, the machine language symbols representing such values are treated as unsigned integers.

Literal values are thus right justified during comparison, though if they are of different length, the shorter literal value is prefixed by blanks. Although the relation between two literal values depends on their numeric encoding, it can usually be interpreted in an alphabetic sense.

Status values are similarly compared on the basis of their numeric encoding. (The status value specified by a file name is one of the set of statuses associated, by the file declaration, with the states of the file. In this context, a file name specifies the current state of the file and may be thought of as a status pseudo-variable, automatically updated prior to the comparison.)

Entry values, finally, are represented by single, composite symbols which are likewise compared as unsigned integers.

Form ∅65. bform ǂ ⁅NOT bform⁆;⁅(bform)⁆;⁅bform AND;OR bform⁆

The logical operators AND, OR, and NOT allow simple Boolean formulas to be combined into more complex Boolean formulas. Their meaning is given in the following table:

p	q	NOT q	p AND q	p OR q
∅	∅	1	∅	∅
∅	1	∅	∅	1
1	∅		∅	1
1	1		1	1

The scope of a logical operation in a Boolean formula may be indicated by the use of parentheses. In the absence of parentheses, NOT's are performed first, then AND's and finally OR's. Within these categories, the sequence of operations is from left to right.

Examples

```
NOT ALPHA+BETA($91$) LS ∅
BOOL AND NOT (ALPHA GR BETA($C**2$) OR ATOM(FORM($C$)))
```

Entry Formula.

Form ∅66. entform ǂ ∅;entvar

The value specified by an entry formula is an entry value denoted by zero or designated by an entry variable.

Examples

```
ø
ENTRY(DATA8($GAMMA$))
```

Sequential Formula.

Form ø67. seqform ⫳ statementname;⸢switchname ℓ;⸢($ index $)⸣⸣

A sequential formula specifies a sequel in the sequence of statement executions. The value specified by a sequential formula is thus a statement name. This value may be directly denoted, or it may be the value designated by a simple or subscripted switch name. A switch name refers to the defining switch declaration which lists the set of statement names that the switch may designate as values. Since these statement names are themselves specified by sequential formulas, the determination of a switch's value is obviously a recursive process.

Two kinds of switches may be declared: index switches; and item switches. The value of the 1-component index subscripting an index switch serves to indicate directly a switch value from the list in the switch declaration. The index (if any) subscripting an item switch serves to index the item whose name is part of the switch declaration. The value so designated is used (in a manner described in the switch declaration) to determine the value of the switch.

In some cases, a switch may have no value, since sequential formulas need not be specified for all possible values of the index or the item. A sequential formula with no value has no effect on the statement execution sequence.

Examples

S∅1

MATRIX'MULTIPLICATION

SWIMAGE(B)

WHICH($DELTA+I,'DELTA+J,DELTA+K$)

SENTENCES

With certain delimiters, expressions form statements and phrases form declarations. These are the sentences of JOVIAL.

STATEMENTS

Statements are the operational units of JOVIAL, and describe closed and self-contained rules of computation. The statements of a program or procedure are normally executed in the sequence listed. However, this sequence of operations may be broken by GOTO statements, which define their successor explicitly; shortened by IF statements, which may cause certain statements to be iterated; and otherwise altered by CLOSE statements, which close or remove certain statements from the normal sequence of statement executions. Statements may be named, and sequences of statements may be grouped into compound statements. Simple (non-compound) statements are invariably terminated by the termination separator $.

Named Statement.

Form ∅68. statement ⫳ statementname . statement

A statement may be identified by attaching a name to it. The name precedes the statement it identifies, which may be simple or compound, or already named, and is separated from it by the statement name separator . which is distinguished from the decimal point by context.

Examples

```
RESET'CELL'COUNT.  CELL'COUNTER = Ø $
HALT. QUIT. CEASE. DESIST. WHOA.  STOP $
CLEAR.  BEGIN FOR R = ALL(BETA) $  BETA($R$) = Ø $ END
```

Compound Statement.

Form Ø69.   statement ⫲ BEGIN $\{$ declaration;statement $\}_0$ END

A string of statements, which may be interspersed with declarations, are combined into a single, compound statement by enclosing them in the statement brackets BEGIN and END. The statements comprising a compound statement may themselves be compound.

Examples

```
BEGIN IF CTR GR Ø $  BEGIN CTR = CTR-1 $  RETURN $ END END
BEGIN MP = (MN+MM($K$))/2 $  MM($K$) = MN $  MN = MP $ END
```

Assignment Statement.

An assignment statement assigns the value specified by a formula to be the value thereafter designated by a variable. The symbol = separating the left-hand variable from the right-hand formula is the assignment separator and, denoting assignment, has an imperative meaning not to be confused with that of the equals sign of ordinary mathematics, which merely states a passive condition of equality

Form Ø7Ø.   statement ⫲ avar = aform $

An arithmetic assignment statement assigns an arithmetic variable the value specified by an arithmetic formula. If the value is specified to greater precision than required by the variable, the excess precision is truncated unless, in the case of an item, rounding (to the closest value with the required precision) is specifically indicated in the item description. If the value is specified to less precision than required by the variable, the remaining bit positions will be filled in with bits of zero magnitude. If the value specified has greater significance than accepted by the variable, the value's most significant bits <u>may</u> be lost during assignment -- a condition known as 'overflow'.

The value specified by a mono-valued formula will be 'twinned' when assigned to a dual item and, where necessary, an automatic conversion between floating-point and fixed-point representation is implied.

The results of assigning a negative value to an unsigned variable, and a dual value to a mono-valued variable, are undefined.

Examples
```
DELTA = -(BETA($X-1$)+1.65E4A-3) ** -5 $
BETA($D$) = BETA($D$)+1 $
BIT($I,K+2$)(DELTA) = (/MAXIMUM(OMEGA, 2Ø.) * GAMMA/) $
```

Form Ø71.   statement ╪ lvar = lform $

Examples
```
TAG = 6H(      ) $
ALPHA($D$) = ALPHA($D+1$) $
BYTE($K/2,8$)(LINE'IMAGE($I,DELTA-J$)) = O(77777777777777777) $
```

Form Ø72.   statement ╪ svar = sform $

Examples
```
WEATHER($K$) = V(GLIM) $
ARGUS($BETA($D$)$) = MODUS $
```

Form Ø73.   statement ╪ bvar = bform $

Examples
```
BOOL = NOT (DELTA+7.4 EQ (/BETA($K$)*3./) OR BOOL) AND SIGN(GAMMA) $
INDIC($DELTA**2$) = 1 $
```

Form Ø74.   statement ⍋ entvar = entform $

Examples

```
ENTRY(BETA($K$)) = Ø $
ENTRY(BETA($K$)) = ENTRY(BETA($J$)) $
```

Non-arithmetic assignment statements operate as if on unsigned integers. This has the effect of right justification on literal values. A literal value assigned to a shorter variable may thus lose its leading signs. A literal value assigned to a longer variable, however, is prefixed by blanks.

Exchange Statement.

Form Ø75.   statement ⍋ avar == avar $

Examples

```
ALPHA == B $
DELTA($I$) == DELTA($I+1$) $
CHAR(GAMMA) == BIT($Ø,CHAR'SIZE$)(R273($A,B,C,D$)) $
```

Form Ø76.   statement ⍋ lvar == lvar $

Examples

```
WORD($Y$) == WORD($Ø$) $
BYTE($U$)(FEW($V$)) == BYTE($W$)(GAMMA) $
```

Form Ø77.   statement ⍋ svar == svar $

Examples

```
ARGUS($GAMMA$) == ARGUS($GAMMA-1$) $
MODE'A == MODE'B $
```

Form Ø78.   statement ⍋ bvar == bvar $

Examples

```
BOOL == MODUS($I$) $
SIGN(DELTA) == INDICATOR($I,J,K$) $
```

Form Ø79. statement ⸗ entvar == entvar $

Examples

```
ENTRY(DART2($Ø$)) == ENTRY(DART2($S$)) $
ENTRY(BETA($K$)) == ENTRY(DELTA($K$)) $
```

An exchange statement exchanges the values designated by a pair of compatible variables. An exchange operates as if the value designated by each variable were assigned, concurrently, to the other. The protocols of the assignment statement are thus doubly applicable. The symbol == separating the pair of variables is the exchange separator.

GOTO Statement.

Form Ø8Ø. statement ⸗ GOTO seqform $

A GOTO statement may interrupt the ordinary, listed sequence of statement executions, defining its successor explicitly by the value specified by a sequential formula. This interruption will not occur if the sequential formula has no value, as may be the case with a switch, and the next statement executed will therefore be the next listed. If the value of the sequential formula is the name of a closed statement, the interruption may only be temporary, since a closed statement, upon execution, will normally return control to the next statement listed after the GOTO statement that activated it. And finally, if the value of the sequential formula is an ordinary statement name, the interruption of the statement execution sequence will be permanent, with the next statement executed being the one bearing the specified statement name.

Examples

```
GOTO SØ1 $
GOTO SLEEP $
GOTO SWIMAGE($A-3+DELTA$) $
```

IF Statement.

Form Ø81. statement ≬ IF bform $

An IF statement causes the evaluation of a Boolean formula. If false, the execution of the next statement listed will be omitted. Thus, the execution of any statement can be made to depend on the truth of a Boolean formula by preceding the statement with an IF statement. Put another way: an IF statement may be said to combine with the next statement listed to form a statement pair with the same operational effect as its latter part if the Boolean formula specifies the value true, but with the effect of 'no operation' otherwise.

Examples

```
IF BOOL $
IF EQUAL(BYTE($U,12$)(WORD), 12H(ESOTROOROISM)) AND SIGN(DELTA) $
IF TEMPERATURE EQ V(HIGH'6Ø'S) $
IF DELTA**2/BETA($C-3$) LQ (/GAMMA/) OR BETA($C$) NQ 18.4 $
```

FOR Statement

The execution of a FOR statement defines a subscript (integer valued variable identified by a single letter) whose scope of definition includes just the FOR-statement string defining it, and the simple or compound statement immediately following. A FOR-statement string, which is a FOR statement and any FOR-statement string immediately following it, causes the statement which it precedes to be repeatedly executed, one or more times. Prior to each such execution, values are assigned to some or all of the subscripts defined by the FOR-statement string. In general, the process consists of the five following steps:

A. Initialize: Assign initial value(s) to the subscript(s).

B. Execute: Execute the statement following.

C. Modify: Assign modified value(s) to the subscript(s).

D. Test: Test the value of the controlling subscript to determine whether or not to omit the next step.

E. Iterate: Return to step B.

The list of arithmetic formulas following the assignment separator in a FOR statement gives a rule for obtaining the sequence of values which are to be assigned to the subscript. The first arithmetic formula specifies the subscript's initial value (step A). The second specifies the increment used to modify the subscript's value (step C), and implies iteration (step E). The third arithmetic formula specifies a final, boundary value used to test the subscript's value (step D). The last one or two arithmetic formulas, and the steps corresponding, may be omitted.

It should be clear that the modify, test, and iterate steps (C, D, and E) are all expressible in terms of more elementary statement forms. In fact, a FOR-statement string functions merely as an abbreviation for these statements. It is therefore important to note that these steps imply a compound statement for their realization. This implied statement, automatically inserted, has all the properties of an equivalent, explicit statement.

Two points concerning the scope of definition of a subscript must be emphasized:

1. A FOR statement may not re-define a subscript already defined.

2. A subscript is defined over a simple or compound statement only if the statement previously executed was the FOR-statement string defining the subscript. This obviously excludes a GOTO statement entering a loop from outside the loop.

Form ∅82. statement ∮ FOR @ = aform $

A 1-factor, incomplete FOR statement, containing a single arithmetic formula, serves only to define a subscript and assign it a value. A FOR-statement string consisting of one or more FOR statements of this type results in a one-pass loop; that is, since no modification, test, or iteration is generated, the statement following is executed only once.

Examples

```
FOR N = -27 $
FOR Z = BETA($N**2-1$)*5 $
```

Form ∅83. statement ∮ FOR @ = aform , aform $

The pair of arithmetic formulas in a 2-factor, incomplete FOR statement specify a theoretically infinite sequence of values for assignment to the defined subscript. Each new value is obtained by adding the value of the second arithmetic formula to the current value of the subscript. A FOR-statement string containing one or more 2-factor FOR statements (and no 3-factor FOR statement) results in a loop with potentially an infinite number of iterations. Since this is clearly intolerable in a correctly functioning program, it is the programmer's responsibility to provide for the ultimate termination of the loop by transferring the execution sequence to a statement outside the loop after a finite number of iterations.

Examples

```
FOR Y = ∅, 2 $
FOR H = ALPHA, -6*BETA($Y$) $
```

Form ∅84. statement ∮ FOR @ = aform , aform , aform $

A 3-factor, complete FOR statement specifies a finite sequence of values for assignment to the defined subscript, each subsequent value being obtained by adding the value specified by the middle arithmetic formula to the current value of the subscript. The subscript's freshly modified value is then tested to determine whether to include it in the sequence of subscript values, or to terminate the loop. The test compares the subscript's modified value with a final, boundary value--the value specified by the third arithmetic formula of the list. The nature of this comparison depends on the sign of the increment value specified by the middle arithmetic formula. If the current value of this increment is negative (less than zero), then the loop is terminated when the subscript's modified value is less than the boundary value. If the current value of the increment is positive, then the loop is terminated when the subscript's modified value is greater than the boundary value.

A FOR-statement string containing a complete FOR statement produces a loop with a finite number of iterations, equal to the number of values assigned to the completely defined, controlling subscript -- excluding, of course, the value which terminates the loop. A FOR-statement string may contain only one complete FOR statement, which must precede any 2-factor FOR statements in the string. The third arithmetic formula in any complete, 3-factor FOR statement which follows any 2 or 3-factor FOR statements will be ignored, thus converting it into a 2-factor FOR statement for all practical purposes. This means that a FOR-statement string can produce just a single loop. Loops within loops must be constructed by embedding the inner loop in the compound statement being iterated by the outer loop.

Examples

```
FOR M = FIRST, NEXT($M$), Ø $
FOR K = Ø, 1, NENT(BETA)-1 $
FOR R = 99, -1, Ø $
```

Form Ø85. statement ⧫ FOR @ = ALL (tablename;itemname) $

In the utilization of complete FOR statements, by far the most common type of loop processes an entire table, with the number of iterations of the loop equal to the number of entries in the table. Where the order of processing is unimportant, such a loop may be elicited by a FOR-ALL statement which, in effect, is an abbreviation of either:

```
FOR @ = Ø,1,NENT(tablename;itemname)-1 $ or
FOR @ = NENT(tablename;itemname)-1,-1,Ø $.
```

Examples

```
FOR S = ALL (BETA)
FOR E = ALL (EMPLOYEE'NUMBER) $
```

TEST Statement.

Form Ø86. statement ⧫ TEST ♀;@ $

A TEST statement serves to terminate the current iteration of a FOR statement loop by transferring the statement execution sequence to one of the implicit subscript modifications at the loop's end. It therefore is defined only when it occurs within a compound statement being iterated by the effect of a FOR-statement string. A TEST statement causes an interruption in the execution sequence similar to that caused by a GOTO statement, but a subscript modification, being only implied, can have no statement name, and thus a GOTO is inapplicable. If a subscript appears in the TEST statement, sequence control is transferred to the modification of that subscript, or if no subscript appears, sequence control is transferred to the first subscript modification of the set.

Subscript modifications are supplied and therefore executed in reverse of the order in which the subscripts were defined in the FOR-statement string, so that, for example, a TEST statement containing the last subscript defined in a FOR-statement string would have the same effect of modifying all the subscripts as would a subscript-less TEST statement. For subscript-controlled FOR-statement loops, the subscript test always follows the subscript modification(s). Since not all loops are subscript-controlled (e.g., 2-factor FOR statement loops), a subscript test is not always performed, so the word TEST is thus a slight misnomer.

Examples

TEST $

TEST D $

CLOSE Statement.

Form ∅87. statement $ CLOSE statementname $ statement

A CLOSE statement allows the simple or compound statement forming its latter part to be removed from the normal, listed sequence of statement executions. The execution of this 'closed' statement may thus only be definitively invoked by a GOTO statement whose sequential formula has as its value the statement name following the CLOSE sequential operator. The normal successor to a closed statement is that statement listed immediately following the invoking GOTO statement.

Example

```
CLOSE  PCR'SORT $  ''CLOSED STATEMENT WHICH SORTS THE PCR TABLE
              BY KEY ITEM, USING THE SHUTTLE EXCHANGE METHOD.''
              BEGIN
              FOR I = Ø,1,NENT(PCR)-2 $
                BEGIN
                IF PCR'KEY($I+1$) LS PCR'KEY($I$) $
                  BEGIN
                  ENTRY(PCR($I$)) == ENTRY(PCR($I+1$)) $
                  FOR J = I,-1 $
                    BEGIN
                    IF J EQ Ø OR PCR'KEY($J$) GQ PCR'KEY($J-1$) $
                      TEST I $
                    ENTRY(PCR($J$)) == ENTRY(PCR($J-1$)) $
                    END
                  END
                END
              END
```

RETURN Statement.

Form Ø88. statement ≙ RETURN $

A RETURN statement indicates an operational end to a procedure or a closed statement, and may thus appear only within a procedure or a closed statement. It serves to terminate the execution of a procedure or a closed statement by transferring the statement execution sequence to the exit routine which automatically follows the last listed statement of the procedure or closed statement. An exit routine, being an implied function, can have no statement name, and the RETURN statement performs for procedures and closed statements much the same service that TEST does for loops.

STOP Statement.

Form Ø89. statement ≙ STOP ⸤;statementname $

A STOP statement serves to halt, or at least delay for an indefinite time, the sequence of statement executions. It usually indicates an operational end to the program in which it appears.

If a 're-start' of the program is performed by some exterior agency, the execution sequence will resume with the next statement listed, or with the statement bearing the specified statement name if one is given in the STOP statement.

Examples

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

Alternative Statement.

Form ∅9∅. statement ‡ IFEITH bform $ statement $\{_0$ ORIF bform $ statement$\}_0$ END

An alternative statement has the effect of selecting for execution from a set of statements that statement associated with the first true Boolean formula in a corresponding set of Boolean formulas. Each of the Boolean formulas, embedded in an IF-like sub-statement between a sequential operator IFEITH or ORIF and a termination separator, is followed by its associated statement. The Boolean formulas are evaluated in turn until one with the value true is encountered, whereupon the statement associated with that formula is executed. Following this, the sequence of statement executions would normally continue with the next statement listed after the alternative statement. The effect of an alternative statement is therefore equivalent to that of the selected statement by itself. Note, however, that an ORIF sub-statement is not a complete statement, grammatically speaking, and consequently cannot be meaningfully combined with a preceding IF or FOR statement.

Examples

```
IFEITH I EQ J $  DIAG = 1 $  ORIF 1 $  DIAG = ∅ $ END
IFEITH ALPHA GR ∅ $  A = ALPHA*2 $  ORIF ALPHA LS ∅ $  A = ALPHA/2 $
ORIF ALPHA EQ ∅ $  A = A+1 $  END
```

Procedure Statement.

Form ∅91. statement ⇕ procedurename ≬;⌈(≬;⌈formula;itemname; tablename;statementname ≬;⌊ , formula;itemname;tablename;statementname⌋⌉ ≬;⌈= variable;itemname;tablename;⌈statementname .⌉ ≬;⌊ , variable; itemname;tablename;⌈statementname .⌉⌋⌉)⌉ $

A procedure statement serves to call for the execution of a procedure which is a closed and self-contained process with a fixed and ordered set of formal parameters, permanently defined by a procedure declaration. In general, a procedure statement consists of a procedure name, a set (possibly empty) of calling parameters, and necessary delimiters. The assignment separator, where it appears, separates the input calling parameters on the left from the output calling parameters on the right. Calling parameters are either values (as specified by input formulas or designated by output variables) or labels (as directly denoted). A procedure statement, by invoking the procedure, may be said to specify a set of values for the calling parameter output variables that depend on the values of the calling parameter input formulas and that involve the calling parameter labels. The execution of the procedure is effected as though all formal parameters listed in the procedure declaration either designated calling parameter values, or were replaced with calling parameter labels. The calling parameters of a procedure statement are placed in one-to-one correspondence with the formal parameters of the procedure declaration, and must agree with them in number. A calling parameter value must further agree in type (in the sense of assignment) with its corresponding formal parameter, and a calling parameter label must agree in usage.

Examples

```
INITIALIZE $
COMPUTE'TAX $
```

```
A19(8T(34Q'7$AR)) $
TRY(BETA($A$)+1, 2.83E-7, MAXIMUM(Ø,ALPHA)**R) $
SUM(=A) $
SIGMA7(= BETA($A-1$), BIT($Ø,CHI($A-1,7$)$)(ALPHA), GAMMA) $
LINK(ALPHA = BETA($XYZ$)) $
G'C'D(BETA($A$)/3, (/DELTA/)**.5 = Y, ARG'ERROR) $
```

DIRECT Statement.

Form Ø92. statement ǂ DIRECT {sign:|;⸢ASSIGN ⸢A(:|;icon:)
= itemname |;⸢($ index $)⸣⸣;⸢itemname |;⸢($ index $)⸣
= A(:|;icon:)⸣ $ ⸣} JOVIAL

A DIRECT statement allows a routine coded in a more direct (i.e., machine-oriented) programming language to be included for execution among the statements of a JOVIAL program. A DIRECT statement consists of the sign-string comprising the direct language routine, enclosed in the brackets DIRECT and JOVIAL. So that such a routine may manipulate item values, it may include a JOVIAL-like ASSIGN statement which assigns the value designated by an item to be the value contained in the accumulator, an undefined machine register -- or vice versa. The accumulator is denoted by the functional modifier A suffixed with a parenthesized integer constant indicating the number of fractional bit positions within the register, usually zero for all but fixed-point arithmetic variables. If the integer is omitted, the register contains a floating-point arithmetic value. The effect of a DIRECT statement, being machine dependent, is undefined.

Example

```
DIRECT
ASSIGN A(Ø) = FIVE'DECIMAL'DIGIT'LITERAL $
       XCA
       PXD
       CAQ    DECBIN,,5
ASSIGN BINARY'INTEGER($X$) = A(18) $
JOVIAL
```

Input-Output and Files

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

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

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

where the R's are records, the components of the vector, and the p's are partition symbols, with an undefined physical representation, which may be interpreted as:

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

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

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

OUTPUT Statement.

Form Ø93. statement ǂ OPEN;SHUT OUTPUT filename ℓ;constant; variable;itemname;ᑭtablename ℓ;ᑭ($ index ℓ;ᑭ... indexᑫ $)ᑫᑫ $

Examples

```
OPEN OUTPUT  NAME'FILE $
OPEN OUTPUT  PERSONNEL'FILE  PERSONNEL'TABLE $
SHUT OUTPUT  PRINTER $
SHUT OUTPUT  DRUMØ3   RAG4($A...A+99$) $
```

Form Ø94. statement ǂ OUTPUT filename constant;variable; itemname;ᑭtablename ℓ;ᑭ($ index ℓ;ᑭ... indexᑫ $)ᑫᑫ $

Examples

```
OUTPUT  NAME'FILE  DE'BUG($ALPHA*K$) $
OUTPUT  SENSE'LITE'5 1 $
OUTPUT  PRINTER 29H(YOU GOOFED AGAIN, DIDN'T YOU.) $
OUTPUT  PERSONNEL'FILE  PERSONNEL'TABLE($K$) $
```

A file may be written by the execution of a sequence of OUTPUT statements. The first statement executed in such a sequence must be an OPEN OUTPUT statement, and the last statement executed, a SHUT OUTPUT statement.

The 'data-name' portion of an OUTPUT statement, following the file name, specifies the record to be written, which may consist of the bits or bytes representing: a single value, denoted by a constant or designated by a variable; the values comprising an array, indicated by an array item name; the values comprising a table, indicated by a table name; the values comprising a table entry, indicated by a table name subscripted by a 1-component entry index; the values comprising a consecutive set of table entries, indicated by a table name subscripted by a pair of 1-dimensional entry indices separated by the continuation separator ... whose values specify the boundary indices of the entry set.

An OPEN OUTPUT statement serves to determine the availability of the file indicated by the file name, for example: whether or not the appropriate external storage device may be connected. If available, the file is activated and prepared for writing (e.g., a file identification may be written.) An OPEN OUTPUT statement need not specify a record to be written, in which case, file position is initialized tc zero. If, however, an output record is specified, the write operation is initiated and file position is set to 1.

An OUTPUT statement initiates a write operation for the specified output record and increments the file position by 1. The sequence of statement executions may continue, concurrently, with the write operation, although the file is 'busy' until the write is successfully terminated, when all the specified bits or bytes are written without the occurrence of any uncorrectable error in the data transmission. In some files, partition symbols and thus file positions are predetermined. Consequently, a write operation started from the end-of-file position would be unsuccessful. In other files, notably tape files, the partition symbols are determined by the write operation itself so that, in effect, the end-of-file partition symbol follows the last record written.

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

INPUT Statement.

Form Ø95. statement ǂ OPEN;SHUT INPUT filename ≬;variable; itemname;⸢tablename ≬;⸢($ index ≬;⸢... index⸣ $)⸣⸣ $

Examples

```
OPEN INPUT  PERSONNEL'FILE $
OPEN INPUT  CLOCK  T $
SHUT INPUT  CLOCK $
SHUT INPUT  CARD  LINE($K$) $
```

Form ∅96.   statement ⟠ INPUT filename variable;itemname;
⸢tablename ℓ;⸢($ index ℓ;⸢... index⸣ $)⸣⸣ $

Examples

```
INPUT  ENTRY'KEY'8 BOOL $
INPUT  TAPE∅7  MATRIX $
INPUT  PERSONNEL'FILE  PERSONNEL'TABLE($I...J$) $
```

A file may be read by the execution of a sequence of INPUT statements. The first statement executed in such a sequence must be an OPEN INPUT statement, and the last statement executed, a SHUT INPUT statement.

The data-name portion of an INPUT statement, following the file name, designates the values which are to be represented by the bits or bytes read from the input record. These may consist of a single value designated by a variable or a set of values indicated in the manner described for OUTPUT statements.

An OPEN INPUT statement serves to determine the availability of the file indicated by the file name (for example: whether the appropriate external storage device is connected; whether the file identification is correct; etc.) If available, the file is activated and prepared for reading. An OPEN INPUT statement need not designate a record to be read, in which case, file position is initialized to zero. If, however, values are designated to be read from an input record, the read operation is initiated and file position is set to 1.

An INPUT statement initiates a read operation transferring data from the input record to represent the designated values, and increments the file position by 1. The sequence of statement executions may continue, concurrently, with the read operation although the file is 'busy' until the read is successfully terminated. This occurs when a partition symbol is encountered, or when all the designated values have been read from the input record. A read operation is unsuccessful when started from the end-of-file position or when uncorrectable errors occur in the data transmission.

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

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

A SHUT statement is defined only for active files, and an OPEN statement is defined only for inactive files. Further, some file pairs may not be active concurrently, for example: two files on the same tape reel. INPUT-OUTPUT statements are defined only for active files, and in general, an active file may be both written and read, and positioned -- if the file characteristics allow. Thus, a read or position with a serial, write-only file such as a printer is undefined. The characteristics of some files, however, also preclude the initiation of a read or write operation when the file is 'busy', thus eliminating the possibility of stacking input-output operations.

PHRASES

Item Description.

The basic operations of data processing concern the manipulation of the values of items. In JOVIAL, these values are designated by name and, where applicable, by index. Other characteristics required for their manipulation, such as storage location and form of representation, are either implicitly derived or need be supplied only once, and not with each designation. All the necessary explicit characteristics of an item's value may be declared by the programmer with an item description.

Floating Item Description.

Form Ø97. description ⬨ F ⟨;R ⟩;⟨fcon ... fcon⟩

The abbreviation F declares a Floating type item; the optional abbreviation R declares that any value assigned to the item be Rounded instead of truncated; the optional pair of floating constants, separated by the continuation separator ... declare an estimated minimum through maximum absolute value range, for possible use in optimizing the machine language program.

Examples

F

F R

F 17.4...14.4E1Ø

F R 1.137E-6...9.34E24

Fixed (Arithmetic) Item Description.

Form ∅98. description ǂ A n S;U ȣ;ʕȣ;+;-:nʔ ȣ;R ȣ;ʕicon;acon ... icon;aconʔ

The abbreviation A declares a fixed type item; the number declares the total number of bits required by the item, including any sign bit; the abbreviation S declares a Signed item; the abbreviation U declares an Unsigned (positive) item; the optional, signed number declares the number of fractional bits in the item -- zero may be omitted for exact integers; the optional abbreviation R declares that any value assigned to the item be Rounded instead of truncated; the optional pair of fixed or integer constants separated by the continuation separator ... declare an estimated minimum through maximum absolute value range, for possible use in optimizing the machine language program.

Examples

```
A  27  S
A  5∅  S      R  1.7E9A∅ ... 1.12E15A∅
A  ∅8  S  ∅3  R
A  17  U      R
A  18  U  26
A  1∅  U  -4     ∅...1E4
```

Dual Item Description.

Form ∅99. description ǂ D n S;U ȣ;ʕȣ;+;-:nʔ ȣ;R ȣ;ʕdcon ... dconʔ

The abbreviation D declares a Dual type item; and the remainder of the description pertains to either component of the item; the number declares the total number of bits required by each component including any sign bit; the abbreviation S declares a Signed component; the abbreviation U declares an Unsigned (positive) component; the optional, signed number declares the number of fractional bits in a component -- zero may be omitted for exact, dual integers; the optional abbreviation R declares that both components of any (dual) value assigned to the item be Rounded instead of truncated; and the optional pair of dual constants separated by the continuation separator ... declare an estimated minimum through maximum absolute value range for possible use in optimizing the machine language program.

Examples

```
D  16  S      R
D  16  S  ø5
D  24  S  18  R  D(13.98A18,27.A18)...D(27.5A18,31.98A18)
D  2ø  U
D  ø4  U  -2     D(8,8)...D(48,48)
D  45  U  5ø  R
```

Literal Item Description.

Form 1øø. description ǂ H;T n

The abbreviation H declares a Hollerith coded literal item; the abbreviation T declares a Transmission-coded literal item; the number declares the number of bytes in the item.

Examples

```
H 1
T 6
T 499
```

Status Item Description.

Form 1ø1. description ǂ S ⟅;n ⟅ status⟆

The abbreviation S declares a Status type item; the optional number declares the total number of bits to be allocated the item, otherwise derived from the number of the item's status values; the sequence of statuses denote the values of the item, which are represented by the sequential numeric values Ø, 1, 2, 3, and so forth. If a number of bits k is declared, the number of statuses should not exceed 2^k.

Examples

S V(FULL) V(EMPTY) V(SOME)

S 4 V(N) V(NNE) V(NE) V(ENE) V(E) V(ESE) V(SE) V(SSE) V(S) V(SSW) V(SW) V(WSW) V(W) V(WNW) V(NW) V(NNW)

Boolean Item Description.

Form 1Ø2. description ǂ B

The abbreviation B declares a Boolean type item.

Example

B

Parameter Set.

Form 1Ø3. parameter ǂ constant;status;(BEGIN (parameter) END)

A parameter set denotes a value, a list of values, or an array of values which serve to indicate the initial values of a simple item, a table item, a string item, or an array item. The dimension of the parameter set should agree with the dimension of the item being initialized. The values denoted by the elements of a parameter set should, of course, be assignable to that item, and although integer, floating, and fixed constants may be intermixed, the elements of a parameter set should otherwise all be of the same type.

Unless declared by a parameter set, the intial value(s) of an item is (are) not preassigned and is (are) therefore undefined.

Examples

98

2.9378E3A4

V(EXACT)

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

DECLARATIONS

Declarations serve to define certain properties of the identifiers occurring within a procedure, a program, or a program system. These constitute a 3-level hierarchy, with procedures belonging to programs and programs comprising program systems. Identifiers may be defined at any of these levels: by a COMPOOL of system declarations at the program system level; and by programmer supplied declarations at the program and procedure level. All identifiers must be defined by declaration, except statement names which are defined by context, and simple item names which may be defined by mode.

Unless defined by COMPOOL declarations, programmer-supplied definitions of item names, table names, and file names must all appear in the program listing before these names may be definitively used.

By scope of definition, identifiers fall into two distinct and non-overlapping categories: (a) statement and switch names; and (b) item, table, procedure, and file names. Within these two categories, an identifier's scope includes just the procedure, program, or program system for which it was defined and excludes component procedures or programs which define identifiers with the same spelling. Consequently, a particular identifier, defined within a program or procedure, may be used for other purposes outside that program or procedure -- or even outside its above category.

ITEM Declaration.

Form 1Ø4. declaration ⧫ ITEM itemname description $

An ITEM declaration, beginning with the declarator ITEM and ending with the termination separator $, declares a simple item, with the item name taking on the type indicated by the item description.

Examples

```
ITEM  DELTA  F  1.283E-7...7.9E16 $
ITEM  IDEX8  A  16  U  R $
ITEM  AK'1S  A  74  S  15  R $
ITEM  TRKXY  D  16  S  5 $
ITEM  LINEØ  H  119 $
ITEM  HEAD   S  V(N) V(NE) V(E) V(SE) V(S) V(SW) V(W) V(NW) $
ITEM  INDIC  B $
```

Parameter ITEM Declaration.

Form 1Ø5. declaration ⧫ ITEM itemname ⁅description P parameter⁆; constant $

A parameter ITEM declaration serves to declare a simple item with a specific initial value. This may be done by following the item description with the abbreviation P (for Parameter) and then a single parameter denoting the desired value. For all parameter items except those of status or Boolean type, however, the item description is somewhat redundant, enough so that both it and the abbreviation P may be omitted. In such cases, the parameter determines the item type, along with other necessary information. Note that an octal constant, a 1, or a ∅ all declare integer valued, fixed-point items, rather than literal or Boolean items.

Examples

```
ITEM  AVERAGE  -1.965 $
ITEM  MASQUE  A 36 U  P O(∅∅∅∅7777776∅∅) $
ITEM  SITE'3  D(+14.85A5,-77.∅A5) $
```

MODE Declaration

Form 1∅6.   declaration ⇟ MODE description $\}_0$;$\{^0$P parameter$\}^0$ $

A MODE declaration serves to declare a normal mode of definition for simple (unsubscripted) item names which are not otherwise defined. An item mode pertains to all such undefined item names whose first listed occurrence follows the MODE declaration. Item names so defined take on the type indicated by the item description and may initially designate the value denoted by the single optional parameter. The effect of a MODE declaration persists until superseded by the subsequent listing of another MODE declaration.

Examples

```
MODE  F  $
MODE  A  16  S  R $
MODE  A  16  S  5  P  +39875.E-3A5 $
MODE  T  6 $
MODE  B $
```

OVERLAY Declaration.

Form 1∅7.   declaration ⇟ OVERLAY itemname;tablename $\}_0$;$\{_0$,;= itemname;tablename$\}_0$ $

An OVERLAY declaration arranges items, tables, and arrays in machine storage by allocating to them blocks of consecutive units of storage space. The storage space allocated tables and arrays consists of undefined units called 'registers'. The storage space allocated items depends on a packing mode: No packing -- storage allocated in register units; Medium packing -- storage allocated in sub-register units; Dense packing -- storage allocated in bit position units.

Starting with a specific but undefined origin, the items, tables, and arrays indicated by the set of names separated by commas and delimited by the equals sign =, are allocated a linear, consecutive block of storage space and, except for packed items which may be rearranged for storage efficiency, the order of allocation is the same as the order of names in the declaration. If more than one such set of names is included in an OVERLAY declaration, each set will be allocated storage beginning at the same origin, thus 'overlaying' the others. Consequently, if a name is separated from its predecessor by a comma, storage allocation will (in general) commence immediately following the storage unit just allocated. If, however, the name is separated from its predecessor by an equals sign =, storage allocation will commence at the common origin.

A name may appear only once in an OVERLAY declaration, but may appear in more than one OVERLAY declaration if logical inconsistencies are avoided. To overlay items in a table, the OVERLAY declaration must appear within the TABLE declaration.

TABLE Declaration.

```
Form 1Ø8.     declaration  ⇟  TABLE ℓ;tablename V;R n ℓ;P;S ℓ;N;M;D $
BEGIN ( ITEM itemname description $ ℓ;parameter ℓ;⁽OVERLAY itemname
ℓ;( ,;= itemname) $⁾) END
```

A TABLE declaration serves to declare an optionally named table; the abbreviation V declares a Variable length table; the abbreviation R declares a Rigid length table; the number declares the number of entries (indexed from ∅ thru n-1) comprising the table -- a maximum in the case of a variable length table; the optional abbreviations P or S declare either a Parallel or Serial entry structure; the optional abbreviations N, M, or D declare either No packing, Medium packing, or Dense packing of the items within an entry.

The composition of an entry is described within the BEGIN and END brackets. The set of ITEM declarations declare the items comprising an entry. In designating the value of a table item, a 1-component index subscripting the item name indicates the entry. A parameter list of length k (less than or equal to the number of entries) following an ITEM declaration denotes a list of k values which the first k successive entries of the item are to initially designate. If an OVERLAY declaration is used in describing entry structure, the item names involved must have been previously declared as part of the entry.

Examples

```
TABLE        R  1∅∅ $
BEGIN
ITEM  ALPHA  A  16  U  R $
ITEM  BETA   T  6 $
ITEM  INDIC  B $
END

TABLE TAV∅   V  1∅24  S  D $
BEGIN
ITEM  TTAG   H  18 $
ITEM  TNMI   B $
ITEM  TNOS   B $
ITEM  TTVD   A  12  U $  BEGIN  9 14 33 18 62 78 113 ∅ 77  END
ITEM  TPRO   F  6.89∅3E-6...2.147E9 $
ITEM  TSTA   S  V(NULL)  V(UNDER)  V(LEVEL)  V(OVER) $
OVERLAY  TPRO = TTAG, TTVD $
END
```

```
TABLE FACTORIALS  R  1Ø  N $
                  BEGIN
ITEM  FACTORIAL   A  48  U $  BEGIN  1 1 2 6 24 12Ø 72Ø 5Ø4Ø
                  4Ø32Ø 36288Ø  END
                  END
```

Defined Entry Structure TABLE Declaration.

Form 1Ø9. declaration ≡ TABLE {;tablename V;R n_1 {;P;S n_2 BEGIN { (ITEM itemname description n_3 n_4 {;N;M;D $)$;(STRING itemname description n_3 n_4 {;N;M;D n_5 n_6 $) {;parameter} END

A TABLE declaration of this type serves to declare an optionally named table with a completely specified entry structure. In the TABLE declaration proper: the abbreviation V declares a Variable length table; the abbreviation R declares a Rigid length table; the number n_1 declares the (maximum) number of entries; the optional abbreviations P or S declare either a Parallel or Serial entry structure; and n_2 declares the number of storage registers constituting an entry.

The composition and structure of an entry is described within the brackets BEGIN and END. The set of ITEM and STRING declarations declare the items comprising an entry, and contain information completely specifying and describing the item packing within an entry. An ITEM declaration declares a table item, which has just one occurrence per entry of the table. A STRING declaration declares a string item, which may have a variable number of occurrences (beads) per entry of the table. In designating the value of a particular bead, a 2-dimensional index must subscript the string item name indicating, first, the bead, and second, the entry.

After either the ITEM or STRING declarator, the item name and item description names and describes the item; and following the item description, n_3 declares the index of the register in the entry and n_4 declares the index of the bit position in the register to be the origin, bit Ø, of (the first occurrence of) the item. The optional abbreviations N, M, or D declare the type of item packing arising from the bit allocation: No packing; Medium packing; or Dense packing. In the STRING declaration only: the number n_5 gives the increment to the index of the next register in the entry containing a bead of the string by declaring the frequency of occurrence of the string item in the entry's registers (i.e., every n_5^{th} register); and the number n_6 gives a packing factor by declaring the number of beads per register. The termination separator $ ends both ITEM and STRING declarations.

As before, a parameter list or 1-dimensional parameter set following an ITEM declaration denotes initial values to be designated by the first successive entries of the table item. A 2-dimensional parameter set following a STRING declaration denotes initial values to be designated by the various beads of the string item -- the parameter lists comprising the set denoting values for the first successive entries of the string, and the individual constants within a list denoting values for the beads within an entry.

Examples

```
TABLE  KEY'REFERENCES  V  1ØØØØ  S  1 $
BEGIN
ITEM   KEY  T  6  Ø  ØØ  M $
ITEM   N'REFERENCES  A  12  U  Ø  36  M $
STRING  REFERENCE  A  24  U  1  ØØ  M  1  2 $
END

TABLE  TRC  R  4  2 $
BEGIN
ITEM   TRCA  A  12  U  R  Ø  Ø4 $
ITEM   TRCB  H  6  1  ØØ $
ITEM   TRCC  A  4  S  Ø  Ø  ØØ $  BEGIN -4 3 3 -7 END
ITEM   TRCD  S  20 V(COND'A)  V(COND'B)  V(COND'C)  V(COND'D)  Ø  15 $
```

Like TABLE Declaration

Form 11Ø. declaration ǂ TABLE tablename:@;# ℓ;(V;R n) ℓ;P;S ℓ;N;M;D L $

A like TABLE declaration serves to declare a table with an entry structure like that of a previously declared and named table. The like table's name is formed by suffixing a numeral or letter to the name of the pattern table, and its items are automatically named with the item names, similarly suffixed, of the pattern table. The composition and structure of a like table's entry is taken as being generated by the declarations describing the pattern table's entry structure, with the difference, of course, of the numeral or letter attached to each item name. As with other forms of TABLE declaration, the abbreviation V declares a Variable length table; the abbreviation R declares a Rigid length table; and the number declares the number of entries. If this information is omitted, the pattern table's specifications for these characteristics are used. Again, the optional abbreviations P or S declare either Parallel or Serial entry structure and the optional abbreviations N, M, or D declare either No packing, Medium packing, or Dense packing of the items in the like table's entry. The abbreviation L immediately preceding the termination separator $ declares the table a Like table.

Examples

```
TABLE  G'TABLEØ  L $
TABLE  TAVØQ  R  1  P  D  L $
TABLE  TAVØY  S  N  L $
```

ARRAY Declaration.

Form 111. declaration ǂ ARRAY itemname (n) description $
(;parameter

An ARRAY declaration serves to declare an item name as designating a multi-dimensional array of values, giving the type and the dimensions of the array item. Each number in the sequence of numbers following the item name declares the number of elements in the corresponding dimension of the array. The first number in the sequence declares the number of rows, the second number declares the number of columns, the third number declares the number of planes, and so on. (In designating a particular value within an array, an index with as many components as the number of the array's dimensions must subscript the array item name.) The item description declares the type and other pertinent characteristics of the array item, and is followed by the termination separator $. An array or sub-array of values for the array to initially designate may be denoted by a parameter set following the declaration. The parameter lists between the innermost BEGIN and END bracket pairs denote rows of values, and are thus indexed by row. The individual elements in a parameter list are indexed by column, and the parameter matrices are indexed by plane, and so on.

Examples

```
ARRAY  TIC'TAC'TOE  3  3  S  V(EMPTY)  V(CROSS)  V(NOUGHT) $
ARRAY  NODE  2Ø  18  54  F  8.ØE-6...3.ØE1Ø $
ARRAY  GRID'A  7  5  B $
BEGIN
  BEGIN  Ø Ø 1 Ø Ø  END
  BEGIN  Ø 1 Ø 1 Ø  END
  BEGIN  1 Ø Ø Ø 1  END
  BEGIN  1 1 1 1 1  END
  BEGIN  1 Ø Ø Ø 1  END
  BEGIN  1 Ø Ø Ø 1  END
  BEGIN  1 Ø Ø Ø 1  END
END
```

SWITCH Declaration.

A SWITCH declaration lists the sequential formulas which specify the set of statement names associated with the switch name.

Form 112. declaration ⧫ SWITCH switchname = ({;seqform {;{ , {;seqform}) $

In an index switch, the sequential formulas are indexed according to their position in the list. Each of the k positions of the list, indexed from left to right from ∅ to k-1, is separated by a comma from the adjacent position(s). A position without a sequential formula implies a 'no-value', equivalent to the name of the statement following the GOTO statement invoking the switch. In designating a particular value of an index switch, a 1-component index must subscript the switch name. A switch value is therefore defined only if the index specified is within the index range (∅ thru k-1) of the list.

Example

```
SWITCH  TOGGLE  =  (BL97, , ST∅1($ALPHA$), , , LOOP, EMIT($I,J$)) $
```

Form 113. declaration ⧫ SWITCH switchname (itemname;filename) = (parameter = seqform {;{ , parameter = seqform}) $

In an item switch, the sequential formulas are each associated with a single parameter. The value of the switch depends on the value of the indicated item, or on the current state of the indicated file. The value of the sequential formula associated with the parameter denoting the value of the item or the state of the file is, therefore, the value specified by the switch. However, if none of the parameters in the SWITCH declaration denote the value of the item or the state of the file, the switch's value is 'no-value', equivalent to the name of the statement following the GOTO statement invoking the switch.

Example

```
SWITCH  WHICH (BETA) = (3H(ARY) = ST34, 3H(∅L9) = FINIS($A/2$),
3H(   ) = S∅1, 3H(ABC) = S∅2, 3H(''') = EXIT, 3H(===) = S∅1,
3H(.$.) = ESSO($A,B,C$), O(777777) = ST∅1, 3H(XXX) = PCR'SORT) $
```

DEFINE Declaration

Form 114. declaration ⟠ DEFINE label '':$\{$sign$\}_0$:'' $ Where $\{$sign$\}_0$ does not include the symbol ''

A DEFINE declaration serves to define a label as being equivalent to the 'quoted' sign-string. This equivalence is established by a process of substituting the sign-string for the label wherever it may subsequently be listed, except within another sign-string (for example: in a comment; in the machine language portion of a DIRECT statement; or in a literal constant) or as the label of a DEFINE declaration. This last exception allows a label to be redefined as being equivalent to another sign-string at some later point in the program listing.

Examples

```
DEFINE  ROW  ''127'' $
DEFINE  AS  '' '' $
DEFINE  IF'EITHER AS ''IFEITH'' $
```

FILE Declaration

Form 115. declaration ⟠ FILE filename H;B n_1 V;R n_2 $\{$ status$\}_0$ label $

A FILE declaration serves to declare either a Hollerith file indicated by the abbreviation H, or a Binary file indicated by the abbreviation B. The number n_1 declares the (estimated maximum) number of records in the file; the abbreviation V declares a Variable record size; the abbreviation R declares a Rigid record size; and n_2 declares the (estimated maximum) number of bits or bytes in a record. The set of statuses correspond serially to the states of the file, which are undefined since they depend on the characteristics of the external storage device containing the file, and on the complexity built into the machine language input-output routines by the JOVIAL compiler. The label preceding the termination separator $ is undefined and serves to indicate a particular external storage device in computer dependent terms. A workable definition of this hardware-name label must include a description of the characteristics of the external storage device it references, and an ordered list of the states pertaining to any file using the device.

Examples

```
FILE  NAME'FILE  B  1ØØ  V  9999  V(UNREADY) V(READY) V(BUSY)
V(END'OF'FILE)  TAPEØ3 $
FILE  PRINTER  H  43  R  12Ø  V(UNREADY) V(READY) V(BUSY)  SYSPRT $
```

Procedure Declaration.

A procedure declaration defines a procedure as a statement, headed by a declaration list, whose execution may only be invoked through the use of a procedure statement or a function call. This statement, constituting the body of the procedure, may operate on certain values designated by a fixed set of formal input parameters, may utilize certain labels denoted by a fixed set of formal parameters, and may produce a fixed set of resulting values designated by formal output parameters.

Form 116. declaration ɸ PROC procedurename δ;ρ(δ;ρlabel δ;ɕ , labelʖʔ δ;ρ= label δ;. δ;ɕ , label δ;.ʖʔ)ʔ $ δ;ɕ declarationʖ statement

In general, a procedure declaration consists of the declarator PROC, a procedure name, a set (possibly empty) of formal parameter labels, and necessary delimiters followed by a declaration list which heads the procedure-body statement. The assignment separator =, where it appears, separates the formal input parameters on the left from the formal output parameters on the right, either or both of which may be lacking in a particular case.

To each formal parameter in the procedure declaration must correspond a calling parameter in the invoking procedure statement or function call. This list of calling parameters may contain: formulas, which specify values to be initially designated by formal input parameters; variables, which are assigned the final values designated by formal output parameters; and certain labels, which (in effect only) replace their corresponding formal input or output parameters at each occurrence throughout the heading declarations and the body statements. Formal parameters thus either designate values or denote labels.

Each formal parameter which is to designate a value must be declared by one of the declarations heading the procedure body as the name of a simple item of appropriate type. It is this item which is assigned the value of the calling parameter input formula before the execution of the procedure, or which designates the value assigned to the calling parameter output variable after the execution of the procedure.

All formal parameters not declared as items must denote calling parameter labels -- table names, array item names, or statement names. A formal output parameter meant to denote a statement name must be distinguished by a dummy suffix consisting of a period, whereas a formal input parameter meant to denote a (closed) statement name is not so distinguished.

Formal parameters which denote calling parameter labels may be used within the procedure declaration wherever the substitution of their corresponding labels would be defined. However, a formal parameter meant to denote a table name or an array item name must also be declared as the name of a table or array within the procedure, to provide the procedure with a fixed definition of the structure of the tables or arrays that the formal parameter is to denote.

Any identifier (except a subscript) already defined for the program at a procedure's listing is similarly defined for that procedure, unless an identical identifier, with an overlapping scope, is declared within the procedure or is one of the procedure's formal parameters. All identifiers declared within the procedure, however, are defined only for the procedure, and have no relationship to identical identifiers outside the procedure. No limitations exist on procedure statements or function calls invoking procedures within procedures. However, procedure declarations themselves may not be nested; that is, a procedure declaration may not appear within another procedure declaration.

For a procedure to specify the value of a function, the procedure declaration must have no formal output parameters. The procedure name itself, declared in the procedure heading as an item, must be considered as the sole output parameter, to which the function's value must be assigned during the execution of the procedure.

The execution of a procedure is terminated after the execution of the last procedure-body statement listed, by a RETURN statement, or by a GOTO statement containing a formal output parameter denoting a statement name. The execution of a GOTO statement containing a formal input parameter denoting a (closed) statement name normally only interrupts the execution of the procedure.

Examples

```
PROC        BETA'SUM $ ''A NO-PARAMETER PROCEDURE TO SET THE ARITHMETIC
                     SUM OF THE FIRST N VALUES OF BETA AS THE VALUE OF
                     THE N+1ST BETA.''
                     BEGIN
                     BETA($NENT(BETA)$) = Ø $
                     FOR N = ALL(BETA) $
                         BETA($NENT(BETA)$) = BETA($NENT(BETA)$)
 + BETA($N$) $       NENT(BETA) = NENT(BETA)+1 $
                     END
```

```
PROC          ARCSIN (SINE) $  ''A FIXED-POINT ARCSIN POLYNOMIAL
                   APPROXIMATION FUNCTION.  MAXIMUM ERROR ABOUT
                   .5E-6.  SEE HASTINGS, PAGE 163.''
ITEM          ARCSIN   A 36 S 3Ø R $
ITEM            SINE   A 36 S 35 R $
ITEM           ALPHA 8 A 36 S 34   $  BEGIN +1.57Ø7963Ø5ØA34
                                            -Ø.2145988Ø16A34
                                            +Ø.Ø889798974A34
                                            -Ø.Ø5Ø1743Ø46A34
                                            +Ø.Ø3Ø891881ØA34
                                            -Ø.Ø17Ø881265A34
                                            +Ø.ØØ667ØØ9Ø1A34
                                            -Ø.ØØ12624911A34  END

PROC    RANDOM (=WHERE) $  ''MULTIPLICATIVE PSEUDO-RANDOM NUMBER
               GENERATOR.''
ITEM    WHERE  A 48 U P 53918218906Ø627261 $
ITEM    TEMP   A 96 U $
               BEGIN
               TEMP = WHERE ** 2 $
               WHERE = BIT($24,48$)(TEMP) $
               END

PROC    REFERENCE'SEARCH (WORD = NUMBER, FAILURE.) $  ''THIS PROCEDURE
               SEARCHES THE KEY'REFERENCE TABLE (GIVEN AS AN EXAMPLE
               OF FORM 1Ø9) FOR AN ENTRY WHOSE KEY IS IDENTICAL TO WORD,
               SETTING THE ENTRY-INDEX IN NUMBER.  IF NO KEY MATCHES
               WORD, THE FAILURE EXIT IS TAKEN.  AFTER A SUCCESSFUL
               RETURN, THE SET REFERENCE($Ø...N'REFERENCES($NUMBER$)-1,
               NUMBER$) IS ASSOCIATED WITH THE KEY WORD.''
ITEM    WORD   T 6 $
ITEM    NUMBER A 14 U $
               BEGIN
               FOR I = Ø, 1.5A1+N'REFERENCES($I$)/2.A1, NENT(KEY)-1 $
                 BEGIN
                 IF WORD EQ KEY($I$) $
                   BEGIN
                   NUMBER = I $  RETURN $
                   END
                 END
               GOTO FAILURE $
               END
```

PROGRAM.

Form 117. program $\Diamond$ START $\{$ declaration;statement$\}_0$ TERM $\{$;statementname$\}_0$ $

A JOVIAL program is a string of declarations and statements enclosed in the brackets START and TERM. If a statement name is provided following the TERM, the first statement in the program's execution sequence is the one bearing that name. Otherwise, it is the first (non-procedure-body) statement listed. The typographic end of the program is indicated by the termination separator $.

```
                         START
''L'AUTOMATON.  A PROGRAM WHICH READS A SEQUENCE OF INPUT SYMBOLS,
DECIDES WHETHER OR NOT THEY FORM A SENTENCE IN LANGUAGE L (DESCRIBED
ON PAGE 9 ), AND WRITES THE SEQUENCE AND THE DECISION AS ITS OUTPUT
MESSAGE.  L'AUTOMATON WILL WRITE A REQUEST FOR INPUT AND STOP.
WHEN INPUT IS READY, COMPUTATION MAY RESUME.''
DEFINE          TO '' '' $
DEFINE        TRUE ''1'' $
DEFINE       RIGID ''R'' $
DEFINE      STATUS ''S'' $
DEFINE     INTEGER '' '' $
DEFINE    IFEITHER ''IFEITH'' $
DEFINE    UNSIGNED ''U'' $
DEFINE  HOLLERITH ''H'' $
DEFINE  PROCEDURE ''PROC'' $
DEFINE FIXED'POINT ''A'' $
DEFINE   N'COLUMNS ''    '' $  ''NUMBER OF SYMBOLS PER RECORD
                   (LQ 12Ø) TO BE SUPPLIED BY PROGRAM'S USER.''
FILE       SYMBOLS HOLLERITH 5ØØ ''RECORDS MAXIMUM'' RIGID
                   ''RECORD SIZE OF'' N'COLUMNS  V(INACTIVE)
                   V(AVAILABLE) V(BUSY) V(ERROR) V(FINISHED)
                   CARD'READER $
FILE       MESSAGE HOLLERITH 5ØØ ''RECORDS MAXIMUM'' RIGID
                   ''RECORD SIZE OF'' 12Ø  V(INACTIVE)
                   V(TRANSMITTED) V(BUSY) V(ERROR) LINE'PRINTER $
ITEM   SYMBOL'LIST HOLLERITH N'COLUMNS $
ITEM        RESULT STATUS  V(SENTENCE) V(PHRASE) V(GARBLE)
                   V(EXCESS) $
ITEM         STATE FIXED'POINT 15 ''BIT'' UNSIGNED INTEGER
                   ''RANGE'' Ø...24575 $
```

```
PROCEDURE L'MACHINE (IN'SYMBOL = OUT'SYMBOL) $  ''THIS PROCEDURE
                    IS A SEQUENTIAL MACHINE WHICH, GIVEN A SYMBOL
                    FROM AN INPUT SEQUENCE, WILL RESPOND WITH AN
                    OUTPUT SYMBOL INDICATING WHETHER THE SEQUENCE
                    OF SYMBOLS INPUT THUS FAR IS A COMPLETE SENTENCE
                    IN LANGUAGE L, A PHRASE OR INCOMPLETE SENTENCE,
                    A NON-SENTENCE OR GARBLE, OR WHETHER THE LENGTH
                    OF THE SEQUENCE IS IN EXCESS OF THE MACHINE'S
                    ABILITY TO DECIDE (ROUGHLY AT LEAST 16ØØØ
                    SYMBOLS.)  INITIAL AND TERMINAL BLANKS ARE
                    IGNORED.''
ITEM      IN'SYMBOL HOLLERITH 1 ''CHARACTER'' $
ITEM     OUT'SYMBOL STATUS  V(SENTENCE) V(PHRASE) V(GARBLE)
                    V(EXCESS) $
ITEM         ACTION FIXED'POINT 6 ''BIT'' UNSIGNED INTEGER $
SWITCH    MECHANISM = ( ''1H( )     1H(()     1H($)     1H())     OTHER''
            ''O(ØØØØØ)''  DECIDE,   CHANGE1,  CHANGE2,  GARBLE,   GARBLE,
''O(ØØØØ1)...O(17776)''  GARBLE,   CHANGE1,  CHANGE2,  CHANGE3,  GARBLE,
            ''O(17777)''  GARBLE,   EXCESS,   CHANGE2,  CHANGE3,  GARBLE,
            ''O(2ØØØØ)''  CHANGE2,  GARBLE,   DECIDE,   GARBLE,   GARBLE,
''O(2ØØØ1)...O(37777)''  GARBLE,   GARBLE,   DECIDE,   CHANGE4,  GARBLE,
            ''O(4ØØØØ)''  DECIDE,   GARBLE,   GARBLE,   GARBLE,   GARBLE,
''O(4ØØØ1)...O(57776)''  GARBLE,   GARBLE,   GARBLE,   CHANGE5,  GARBLE,
            ''O(57777)''  EXIT,     EXIT,     EXIT,     EXIT,     GARBLE ) $
                    BEGIN
                      IFEITHER              STATE EQ O(ØØØØØ) $  ACTION = Ø4 $
                      ORIF O(ØØØØ1) LQ STATE LQ O(17776) $  ACTION = Ø9 $
                      ORIF                  STATE EQ O(17777) $  ACTION = 14 $
                      ORIF                  STATE EQ O(2ØØØØ) $  ACTION = 19 $
                      ORIF O(2ØØØ1) LQ STATE LQ O(37777) $  ACTION = 24 $
                      ORIF                  STATE EQ O(4ØØØØ) $  ACTION = 29 $
                      ORIF O(4ØØØ1) LQ STATE LQ O(57776) $  ACTION = 34 $
                      ORIF                  STATE EQ O(57777) $  ACTION = 39 $
                      ORIF TRUE $
                        STOP $  ''PROGRAM OR COMPUTER ERROR.  STATE CAN'T
                        EXCEED O(57777).''
                      END
                      IFEITHER IN'SYMBOL EQ 1H( ) $  ACTION = ACTION - 4 $
                      ORIF     IN'SYMBOL EQ 1H(() $  ACTION = ACTION - 3 $
                      ORIF     IN'SYMBOL EQ 1H($) $  ACTION = ACTION - 2 $
                      ORIF     IN'SYMBOL EQ 1H()) $  ACTION = ACTION - 1 $
                      END
```

```
                  GOTO MECHANISM($ACTION$) $
         GARBLE.  OUT'SYMBOL = V(GARBLE) $
                  GOTO S1 $
         EXCESS.  OUT'SYMBOL = V(EXCESS) $
             S1.  STATE = O(57777) $
           EXIT.  RETURN $
        CHANGE1.  STATE = STATE + O(ØØØØ1) $
                  GOTO DECIDE $
        CHANGE2.  STATE = STATE + O(2ØØØØ) $
                  GOTO DECIDE $
        CHANGE3.  STATE = STATE + O(37777) $
                  GOTO DECIDE $
        CHANGE4.  STATE = STATE + O(17777) $
                  GOTO DECIDE $
        CHANGE5.  STATE = STATE - O(ØØØØ1) $
         DECIDE.    IFEITHER BIT($2,13$)(STATE) EQ Ø $
                      OUT'SYMBOL = V(SENTENCE) $
                    ORIF TRUE $
                      OUT'SYMBOL = V(PHRASE) $
                    END
                  END
    L'AUTOMATON.  BEGIN
                  OPEN OUTPUT MESSAGE 46H(L'AUTOMATON READY FOR
INPUT SEQUENCE. PROCEED.) $
                  STATE = O(ØØØØØ) $
                  STOP $
                  OPEN INPUT SYMBOLS $
  READ'SYMBOLS.   INPUT SYMBOLS TO SYMBOL'LIST $
    READY'TEST.   IF SYMBOLS EQ V(AVAILABLE) AND MESSAGE EQ
V(TRANSMITTED) $
                    BEGIN
                    OUTPUT MESSAGE SYMBOL'LIST $
                    FOR I = Ø, 1, N'COLUMNS - 1 $
                      L'MACHINE(BYTE($I$)(SYMBOL'LIST), RESULT) $
                    GOTO READ'SYMBOLS $
                    END
                   IF SYMBOLS EQ V(BUSY) OR MESSAGE EQ V(BUSY) $
                    GOTO READY'TEST $
                    IF SYMBOLS EQ V(FINISHED) $
                      BEGIN
                      SHUT INPUT SYMBOLS $
                        IFEITHER RESULT EQ V(SENTENCE) $
                          SHUT OUTPUT MESSAGE 47H(THE ABOVE SEQUENCE
IS A SENTENCE IN LANGUAGE L.) $
                        ORIF RESULT EQ V(PHRASE) $
                          SHUT OUTPUT MESSAGE 45H(THE ABOVE SEQUENCE
IS A PHRASE IN LANGUAGE L.) $
                        ORIF RESULT EQ V(GARBLE) $
                          SHUT OUTPUT MESSAGE 45H(THE ABOVE SEQUENCE
IS A GARBLE IN LANGUAGE L.) $
                        ORIF RESULT EQ V(EXCESS) $
                          SHUT OUTPUT MESSAGE 73H(THE LENGTH OF THE
ABOVE SEQUENCE EXCEEDS L'AUTOMATON'S ABILITY TO DECIDE.) $
                        END
                      GOTO L'AUTOMATON $
                      END
                    STOP L'AUTOMATON $
                    END
                    TERM L'AUTOMATON $
```