(2) W. Keister, A. E. Ritchie, S. H. Washburn; "The Design of Switching Circuits", D. Van Nostrand, N. Y. 1950, page 485.
(3) A. D. Booth, "Automatic Digital Calculators," Butterworth's Scientific Pub., London, 1953, page 173.
(4) P. K. Richards, "Arithmetic Operations in Digital Computers," D. Van Nostrand, N. Y. 1955, page 293.
(5) P. A. MacMahon, "Combinatory Analysis," Cambridge University, 1915, page 3.

Editor's Notes:

Although this method is not novel, it has been printed here to summarize for the benefit of a new generation of computer personnel. It should be noted that:

1) This method seems advantageous if only a few significant figures are required. Otherwise the normal method, Log-Multiply-Antilog, is more desirable and faster in particular for higher order roots. These subroutines are normally required for other purposes anyway and space is not lost.
2) One immediately notices many tricky ways of coding this method for a computer, via looping and the use of tables or converting instructions. Note that, as one proceeds, the contribution of the left-hand term becomes proportionately large enough such that it alone might be used within accuracy limits after a certain number of digits are developed.
3) Although the author states that this method used more memory space than other routines, it seems that the converse could well be true if advantage were taken of higher order differences in building up the subtrahend. This appears to be a natural method for a 256 memory machine, if it had good indexing and looping features. Remember that $\Delta^n(X^n) = $ a constant n!

PRELIMINARY REPORT—INTERNATIONAL ALGEBRAIC LANGUAGE

Note:

In the interest of immediate circulation of the results of the ACM-GAMM committee work on an algebraic programming language, this preliminary report is presented. The language described naturally enough represents a compromise, but one based more upon differences of taste than on content or fundamental ideas. Even so, it provides a natural and simple medium for the expression of a large class of algorithms. This report has not been thoroughly examined for errors and inconsistencies. It is anticipated that the committee will prepare a more complete description of the language for publication.

Acknowledgements:

The members of the conference wish to express their appreciation to the Association for Computing Machinery, the "Deutsche Forschungsgemeinshaft" and the Swiss Federal Institute of Technology for substantial help in making this conference and resultant report possible.

A. J. PERLIS
K. SAMELSON
for the ACM-GAMM Committee

PART I—INTRODUCTION

In 1955, as a result of the Darmstadt meeting on electronic computers, the GAMM (association for applied mathematics and mechanics), Germany, set up a committee on programming (Programmierungsausschus). Later a subcommittee began to work on formula translation and on the construction of a translator, and a considerable amount of work was done in this direction.

A conference attended by representatives of the USE, SHARE, and DUO organizations and the Association for Computing Machinery (ACM) was held in Los Angeles on 9 and 10 May 1957 for the purpose

of examining ways and means for facilitating exchange of all types of computing information. Among other things, these conferees felt that a single universal computer language would be very desirable. Indeed, the successful exchange of programs within various organizations such as USE and SHARE had proved to be very valuable to computer installations. They accordingly recommended that the ACM appoint a committee to study and recommend action toward a universal programming language.

By Oct 1957 the GAMM group, aware of the existence of many programming languages, concluded that rather than present still another formula language, an effort should be made toward unification. Consequently, on 19 Oct 1957, a letter was written to Prof. John W. Carr III, president of the ACM. The letter suggested that a joint conference of representatives of the GAMM and ACM be held in order to fix upon a common formula language in the form of a recommendation.

An ACM Ad-Hoc committee was then established by Dr. Carr, which represented computer users, computer manufacturers, and universities. This committee held three meetings starting on 24 Jan 1958 and discussed many technical details of programming language. The language that evolved from these meetings was oriented more towards problem language than toward computer language and was based on several existing programming systems. On 18 April 1958 the committee appointed a subcommittee to prepare a report giving the technical specifications of a proposed language.

A comparison of the ACM committee proposal with a similar proposal prepared by the GAMM group (presented at the above-mentioned ACM Ad-Hoc committee meeting of 18 April 1958) indicated many common features. Indeed, the GAMM group had planned on its own initiative to use English words wherever needed. The GAMM proposal represented a great deal of work in its planning and the proposed language was expected to find wide acceptance. On the other hand the ACM proposal was based on experience with several successful, working problem oriented languages.

Both the GAMM and ACM committees felt that because of the similarities of their proposals there was an excellent opportunity for arriving at a unified language. They felt that a joint working session would be very profitable and accordingly arranged for a conference in Switzerland to be attended by four members from the GAMM group and four members from the ACM committee. The meeting was held in Zurich, Switzerland, from 27 May to 2 June 1958 and attended by F. L. Bauer, H. Bottenbruch, H. Rutishauser, and K. Samelson from the GAMM committee and by J. W. Backus, C. Katz, A. J. Perlis, and J. H. Wegstein for the ACM Committee.[1]

It was agreed that the contents of the two proposals should form the agenda of the meeting, and the following objectives were agreed upon:

I. The new language should be as close as possible to standard mathematical notation and be readable with little further explanation.

II. It should be possible to use it for the description of computing processes in publications.

III. The new language should be mechanically translatable into machine programs.

There are certain differences between the language used in publications and a language directly usable by a computer. Indeed, there are many differences between the sets of characters usable by various computers. Therefore, it was decided to focus attention on three different levels of language, namely a *Reference Language*, a *Publication Language*, and several *Hardware Representations*.

Reference Language

1. It is the working language of this committee.
2. It is the defining language.
3. It has only one unique set of characters.

[1] In addition to the members of the conference, the following people participated in the preliminary work of these committees:
GAMM P. Graeff, P. Lauchli, M. Paul, F. Penzlin
ACM D. Arden, J. McCarthy, R. Rich, R. Goodman, W. Turanski, S. Rosen, P. Desilets, S. Gorn, H. Huskey, A. Orden, D. C. Evans

4. The characters are determined by ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.
5. It is the basic reference and guide for compiler builders.
6. It is the guide for all hardware representations.
7. *It will not normally be used stating problems.*
8. It is the guide for transliterating from publication language to any locally appropriate hardware representations.
9. The main publication of the common language itself will use the reference representation.

Publication Language (see Part IIIc)

1. The description of this language is in the form of permissible variations of the reference language (e.g., subscripts, spaces, exponents, Greek letters) according to usage of printing and handwriting.
2. It is used for stating and communicating problems.
3. The characters to be used may be different in different countries but univocal correspondence with reference representation must be secured.

Hardware Representations

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.
2. Each one of these uses the character set of a particular computer and is the language accepted by a translator for that computer.
3. Each one of these must be accompanied by a special set of rules for transliterating from Publication language.

PART II—DESCRIPTION OF THE REFERENCE LANGUAGE

A. *STRUCTURE OF THE LANGUAGE*

As stated in the introduction, the algorithmic language has three different kinds of representations—reference, hardware, and publication and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols—and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well-known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language—explicit formulae—called arithmetic statements.

To show the flow of larger computational processes, certain non-arithmetic statements are added which may describe alternatives or recursive repetitions of computing statements.

Statements may be supported by declarations which are not themselves computing rules, but inform the translator of certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers or even the set of rules defining a function.

Sequences of statements and declarations, when appropriately combined, are called programs. However, whereas complete and rigid formal rules for constructing translatable statements are described in the following, no such rules can be given in the case of programs. Consequently, the notion of program must be considered to be informal and intuitive, and the question whether a sequence of statements may be called a program should be decided on the basis of the operational meaning of the sequence.

In the sequel explicit rules—and associated interpretations—will be given describing the syntax of the language. Any sequence of symbols to which these rules do not assign a specific interpretation will be considered to be undefined. Specific translators may give such sequences different interpretations.

B. *BASIC SYMBOLS*

The reference language is built up from the basic symbols listed in Part IIIa. These are:

1. *Letters* λ (the standard alphabet of small and capital letters)
2. *Figures* ζ (arabic numerals 0, ... 9)
3. *Delimiters* δ consisting of
   a. operators ω:
      arithmetic $+ \quad - \quad \times \quad /$
      relational $< \quad \leq \quad = \quad \geq \quad > \quad \neq$
      logical $\neg \quad \vee \quad \wedge \quad \equiv$
      sequential *go to do return stop for if or if either or if*
   b. separators σ: . , : ; := =: $\rightarrow$ $_{10}$ *begin end*
   c. brackets β: ( ) [ ] $\uparrow$ $\downarrow$
   d. declarators ϕ: *procedure array switch type comment*

Of these symbols, letters do not have individual meaning. Figures and delimiters have a fixed meaning which for the most part is obvious, or else will be given at the appropriate place in the sequel.

Strings of letters and figures enclosed by delimiters represent new entities. However, only two types of such strings are admissible:

1. Strings consisting of figures ζ only represent the (*positive*) *integers* G (including 0) with the conventional meaning.
2. Strings beginning with a letter λ followed by arbitrary letters λ and/or figures ζ are called *identifiers*. They have no inherent meaning, but serve for identifying purposes only.

C. *EXPRESSIONS*

Arithmetic and logical processes (in the most general sense), which the algorithmic language is primarily intended to describe, are given by arithmetic and logical expressions, respectively. Constituents of these expressions, except for certain delimiters, are numbers, variables, elementary arithmetic operators and relations, and other operators called functions. Since the description of both variables and functions may contain expressions, the definition of expressions, and their constituents, is necessarily recursive.

The following are the units from which expressions are constructed:

1. (*positive*) *Numbers N*
   Form: $N \sim G.G_{10} \pm G$ where each G is an integer as defined above.
   G.G is a decimal number of conventional form. The scale factor $_{10} \pm G$ is the power of ten given by $\pm G$. The following constituents of a number may be omitted in any occurrence: The fractional part .00 . . . 0 of integer decimal numbers; the integer 1 in front of a scale factor; the + sign in the scale factor; the scale factor $_{10} \pm 0$.
   Examples: 4711 137.06 $2.9997_{10}10$ $_{10}{}^{-}12$ $3_{10}{}^{-}12$

2. *Simple Variables V*
   are designations for arbitrary scalar quantities; e.g., numbers as in elementary arithmetic.
   Form: $V \sim I$ where I is an identifier as defined above.
   Examples: a X11 PSI2 ALPHA

3. *Subscripted Variables V*

designate quantities which are components of multidimensional arrays.

Form: V ~ I [C]

where C ~ E, E, ~~~~, E is a *list* of arithmetic expressions as defined below. Each expression E occupies one subscript position of the subscripted variable, and is called a *subscript.* The complete list of subscripts is enclosed in the subscript brackets [ ].

The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (cf. arithmetic expressions).

Subscripts, however, are intrinsically integer-valued, and whenever the value of a subscript expression is not integral, it is replaced by the nearest integer (in the sense of proper round off).

Variables (both simple and subscripted ones) designate arbitrary real numbers unless otherwise specified. However, certain declarations (cf. *type declarations*) may specify them to be of a special type, e.g., *integral,* or *Boolean.* Boolean (or logical) variables may assume only the two values *true* and *false.*

4. *Functions F*

represent single numbers (function values), which result through the application of given sets of rules to fixed sets of parameters.

Form: F ~ I (P, P, ~~~~, P)

where I is an identifier, and P, P, ~~~~, P is the ordered list of actual parameters specifying the parameter values for which the function is to be evaluated. A syntactic definition of parameters is given in the sections on *function declarations* and *procedure declarations.* If the function is defined by a *function declaration,* the parameters employed in any use of the function are expressions compatible with the type of variables contained in the corresponding parameter positions in the function declaration heading (cf. *function declaration*). Admissible parameters for functions defined by *procedure declarations* are the same as admissible input parameters of procedures as listed in the section on *procedure statements.*

Identifiers designating functions, just as in the case of variables, may be chosen according to taste. However, certain identifiers should be reserved for the standard functions of analysis.

This reserved list should contain:

abs (E)	for the modulus (absolute value) of the value of the expression E
sign (E)	for the sign of the value of E
entire (E)	for the largest integer not greater than the value of E
sqrt (E)	for the square root of the value of E
sin (E)	for the sine of the value of E

and so on according to common mathematical notation.

5. *Arithmetic expressions E* are defined as follows:

a. A number, a variable (other than Boolean), or a function is an expression.

Form: E ~ N ~ V ~ F

b. If E_1 and E_2 are expressions, the first symbols of which are neither "+" nor "−", then the following are expressions:

E ~ + E_1	~ $E_1 \times E_2$
~ − E_2	~ E_1 / E_2
~ E_1 + E_2	~ $E_1 \uparrow E_2 \downarrow$
~ E_1 − E_2	~ (E_1)

The operators +, −, ×, / appearing above have the conventional meaning. The parentheses $\uparrow \downarrow$ denote exponentiation, where the leading expression is the base and the expression enclosed in parentheses is the exponent.

Examples: $2 \uparrow 2 \uparrow n \downarrow \downarrow$ means $2^{(2^n)}$

$2 \uparrow 2 \downarrow \uparrow n \downarrow$ means $(2^2)^n$

$a \uparrow 2 \uparrow b \downarrow \downarrow \uparrow 2 \downarrow$ means $\left(a^{2^b}\right)^2$

The proper interpretation of expressions can always be arranged by appropriate positioning of parentheses. An arithmetic expression is a rule for computing one real number by executing the indicated arithmetic operations on the actual numerical values of the constituents of the expression. This value is obvious in the case of numbers N. For variables V, it is the current value (assigned last in the dynamic sense), and for functions F it is the value arising from the computing rules defining the function (cf. *function declaration*) when applied to the current values of the function parameters given in the expression.

The sequence of operations within one expression is generally from left to right, with the following additional rules:

a. parentheses are evaluated separately

b. for operators, the conventional rule of precedence applies:
first: $\times$ / second: + −

In order to avoid misunderstandings, redundant parentheses should be used to express, for example, $\frac{ab}{c}$ in the form (a×b)/c or (a/c)×b rather than by a×b/c, or a/c×b, respectively, and to avoid constructions such as a/b/c.

Examples:	A	A [j+k−2, j−k]
	Alpha	A [mu [s]]
	Degree	a × sin (omega × t)
	A [1, 1]	0.5 × a [(N × (N−1))/2, 0]

6. *Boolean expressions* B are defined analogously to arithmetic expressions:

a. A truth value, a variable (Boolean by declaration), or a function (Boolean by declaration) is an expression.
Form: $B \sim 0$ (the truth value *false*) $\sim V$
~ 1 (the truth value *true*) $\sim F$

b. If E_1 and E_2 are arithmetic expressions, then the following arithmetic relations are expressions:
$B \sim (E_1 < E_2)$ $\sim (E_1 \neq E_2)$ $\sim (E_1 > E_2)$
$\sim (E_1 \leq E_2)$ $\sim (E_1 \geq E_2)$ $\sim (E_1 = E_2)$
Such expressions take on the (current) value *true* whenever the corresponding relation is satisfied for the expression involved, otherwise *false*.

c. If B_1 and B_2 are expressions, the following are expressions:
$B \sim \neg B_1$ $\sim B_1 \equiv B_2$
$\sim B_1 \vee B_2$ $\sim (B_1)$
$\sim B_1 \wedge B_2$
The operators $\neg$, $\vee$, $\wedge$, $\equiv$ have the interpretations *not*, *or*, *and*, and *equivalent*. Interpretation of the binary operators will be from left to right. The scope of "$\neg$" is the first expression to its right. Any other desired precedence must be indicated by the use of parentheses.
Examples: $(X = 0)$ $(X > 0) \vee (y > 0)$ $(p \wedge q) \vee (x \neq y)$

D. *STATEMENTS* Σ

Closed and self-contained rules of operations are called basic Statements Σ. They are defined recursively in the following way:

Strings of one or more statements[2] may be combined into a single (compound) statement by en-

[2] Declarations, which may be interspersed between statements, have no operational (dynamic) meaning. Therefore, they have no significance in the definition of compound statements.

closing them within the "statement parentheses" *begin* and *end*. Single statements are separated by the statement separator ";".

Form: $\Sigma \sim$ *begin* $\Sigma; \Sigma;$ ⌇⌇⌇⌇⌇; Σ *end*

A statement may be made identifiable by attaching to it a label L, which is an identifier I, or an integer G (with the meaning of identifier). The label precedes the statement labeled, and is separated from it by the separator colon (:). Label and statement together constitute a statement called "labeled statement."

Form: $\Sigma \sim L: \Sigma$

A labeled statement may not itself be labeled. In the case of labeled compound statements, the closing parenthesis *end* may be followed by the statement label (followed by the statement separator) in order to indicate the range of the compound statement:

Form: $\Sigma \sim L:$ *begin* $\Sigma; \Sigma;$ ⌇⌇⌇⌇⌇; Σ *end* L;

1. *Assignment statements*

   serve for assigning the value of an expression to a variable.

   Form i) $\Sigma \sim V := E$

   If the expression on the right hand side of the assignment delimiter ":=" is arithmetical, the variable V on the left hand side must also be numerical; i.e., it must not be Boolean. Generally, the arithmetic type of the expression E is determined by the constituents and operations of the expression E. However, V may be declared to be of a special type provided this declaration is compatible with the possible values of the expression E.

   Form ii) $\Sigma \sim V := B$

   If the expression on the right hand side of the assignment statement is Boolean, V may be any variable. This means that the truth values *true* and *false* of the Boolean expression may be interpreted arithmetically as integers "1", and "0", which may then be assigned to a numerical variable.

2. *Go to statements*

   Normally, the sequence of operations (described by the statement of a program) coincides with the physical sequence of statements. This normal sequence of execution may be interrupted by the use of *go to* statements.

   Form: $\Sigma \sim$ *go to* D

   D is a *designational* expression specifying the label of the statement to be executed next. It is either a label L or a switch variable I [E] (cf. *switch declaration*), where I is an identifier and E a subscript expression. In the latter case, the numerical value of E (the value of a subscript) is an ordinal which identifies the component of the switch I (named by declaration). This element which is again a designational expression specifies the label to be used in the *go to* statement. This label determination is obviously a recursive process, since the elements of the switch may again be switch variables.

   Examples: *go to* HADES

   go to exit [(i ↑ 2 ↓ −j ↑ 2 ↓ +I)/2], where exit refers to the declaration

   switch exit := (D_1, D_2, ⌇⌇⌇⌇⌇, D_n)

3. *If statements*

   The execution of a statement may be made to depend upon a certain condition which is imposed by preceding the statement in question by an *if* statement.

   Form: $\Sigma \sim$ *if* B where B is a Boolean expression

   If the value of B is *true*, the statement following the *if* statement will be executed. Otherwise, it will be bypassed and operation will be resumed with the next statement following.

Example: In the sequence of statements
if (a> 0); c:=a ↑ 2 ↓ × b ↑ 2 ↓
if (a <0); c:=a ↑ 2 ↓ + b ↑ 2 ↓
if (a=0); *go to* bed
one and only one of the three statements rightmost in each line will be executed.

4. *For statements*

Recursive processes may be initiated by use of a *for* statement, which causes the following statement to be executed several times, once for each of a series of values assigned to the recursing variable contained in the *for* statement.

Form: i) *for* V := C
ii) *for* V := E_{i_1} (E_{s_1}) E_{e_1}, ⌇⌇⌇, E_{i_k} (E_{s_k}) E_{e_k}
where C is a list of k expressions E_1, E_2, ⌇⌇⌇, E_k; and E_{i_j}, E_{s_j}, E_{e_j} are expressions.

In the Form (i) the intent is to assign to V in succession the value of each expression of the list (expressions taken in order of listing) and the statement following the *for* statement is executed immediately following each such assignment.

In Form (ii) each group of expressions E_i (E_s) E_e determines an arithmetic progression. The value of E_i is the (i)nitial value, E_s gives the value of the increment or (s)tep, and E_e determines the (e)nd value which is the last term of the progression contained in the interval [E_i, E_e]. The intent is to assign to V each value of every progression (these again taken in the order of listing from left to right), and the statement following the *for* statement is executed immediately following each such assignment.

The effect of a *for* statement may be precisely described in terms of "more elementary" statement forms. Thus Form (i) is precisely equivalent to:

V := E_1; Σ; V := E_2; Σ; ⌇⌇⌇; V := E_k; Σ; where Σ is the statement following the *for* statement.

Form (ii) is precisely equivalent to:

V := E_{i_1}; L_1: Σ[3]; V := V+E_{s_1}; *if* (V ≤ [4]E_{e_1}); *go to* L_1; ⌇⌇⌇;
V := E_{i_k}; L_k: *do* L_1; V := V+E_{s_k}; *if* (V ≤ [4]E_{e_k}); *go to* L_k; where Σ is the statement following the *for* statement.

Examples: a) *for* I := 1(1)n; p := p × y + A [I]
b) *for* a := 1, 3, 5, 9.76, ⌇⌇⌇, −13.75;
begin----------
----------*end*

5. *Alternative statements*

An alternative statement is one which has the effect of selecting for execution one from a set of given statements in accordance with certain conditions which exist when the statement is encountered.

Form: *if either* B_1; Σ_1; *or if* B_2; Σ_2; ⌇⌇⌇; *or if* B_k; Σ_k *end* where Σ_i is any statement other than a quantifier; i.e., *if*, *for*, or *or if*, and B_i is any Boolean expression.

The effect of an alternative statement may be precisely described in terms of "more elementary" statement forms. Thus the above form is precisely equivalent to the sequence of statements: *if* B_1; *begin* Σ_1; *go to* next *end*; *if* B_2; *begin* Σ_2; *go to* next *end*; ⌇⌇⌇; *if* B_k; Σ_k where "next" is the label of the statement following the alternative statement.

Example: *if either* (a> 0); y := a+2; *or if* (a <0); y := a/2; *or if* (a=0); y := .57 end.

[3] If Σ is a labeled statement, L_1 is that label. If not, the effect is as though it had a (unique) label L_1. L_k ($k \neq 1$) are theoretically unique labels.

[4] This relational form obtains if the progression is increasing; if decreasing, the relation ≥ is understood to employed.

6. *Do statements*

A statement, or string of statements, once written down, may be entered again (in the sense of copying) in any place of the same program by employing a *do* statement which during copying permits substitution for certain constituents of the statements reused.

Form: $\Sigma \sim do\ L_1, L_2\ (S_{\rightarrow}\rightarrow I, \sim\sim\sim, S_{\rightarrow}\rightarrow I)$

where L_1 and L_2 are labels, the $S_{\rightarrow}$ are strings of symbols not containing the separator ($\rightarrow$) and the I are identifiers, or labels, and the list enclosed by parentheses is a substitution list. The *do* statement operates on the string of statements from, and including, the one labeled L_1 through the one labeled L_2, which statements constitute the range of the *do* statement. If L_1 is equal to L_2; i.e., if the range is just the one statement L_1, the characters "$,L_2$" may be omitted.

The *do* statement causes itself to be replaced by a copy of the string of statements constituting its range. However, in this copy all identifiers or labels, listed on the right hand side of a separator "$\rightarrow$" in the substitution list of the *do* statement, (and which are utilized in these statements) are replaced by the corresponding strings of symbols $S_{\rightarrow}$ on the left hand side of the separators "$\rightarrow$". These string $S_{\rightarrow}$ may be chosen freely with the one restriction that the substitutions produce formally correct statements in the copy[5].

Whenever a *do* statement contains in its range another *do* statement, the copying and substituting process for this second innermost *do* will be executed first. Therefore the (actual) copy induced from a *do* statement never contains a *do* statement. Declarations within the range of a *do* statement are not reproduced in the copy.

Examples: *do* 5, 12 (x [i]→y, black label→red label, ~~~, f (x, y)→g)
do 12A, ABC (x ↑ 2 ↓ +3/y→A, ~~~)

The range of a *do* statement should contain complete statements only; i.e., if the *begin* (*end*) delimiter of a compound statement lies in the range of the *do*, then so should the matching *end* (*begin*). If this rule is not complied with the result of the *do* statement may not be the one desired.

7. *Stop statements*

Stop is a delimiter which indicates an operational (dynamic) end of the program containing it. Operationally, it has no successor statement in that program.

Form: $\Sigma \sim stop$

8. *Return statements*

Return is a delimiter which indicates an operational end of a *procedure*. It may appear only in a *procedure declaration*. (cf. *procedure declaration*.)

Form: $\Sigma \sim return$

9. *Procedure statements*

A *procedure* statement serves to initiate (call for) the execution of a *procedure*, which is a closed and self-contained process with a fixed ordered set of input and output parameters, permanently defined by a *procedure declaration*. (cf. *procedure declaration*.)

Form: $\Sigma \sim I\ (P_i, P_i, \sim\sim\sim, P_i) =: (P_o, P_o, \sim\sim\sim, P_o)$

Here I is an identifier which is the name of some procedure, i.e., appears in the heading of some procedure declaration. (cf. *procedure declaration*.) $P_i, P_i, \sim\sim\sim$ is the ordered list of actual input parameters specifying the input quantities to be processed by the procedure. $P_o, P_o, \sim\sim\sim$ is the ordered list of actual output parameters specifying the variables to which the results of the procedure will be assigned and alternate exits, if any. The procedure declaration defining the called procedure contains, in its heading, a string of symbols identical in form to the procedure statement, and the formal parameters occupying input and output parameter positions there give complete information concerning the admissibility of parameters used in any procedure call, shown by these replacement rules:

[5] Thus, in the copy produced, any designational expression whose range is a statement within the range of the *do* statement must be transformed so that its range refers to the copy produced.

	formal parameters in *procedure* declaration	admissible parameters in *procedure* statement
input parameters	single identifier (formal variable)	any expression (compatible with the type of the formal variable)
	array; i.e., subscripted variable with k = 1 empty parameter positions	array with n(≥k) parameter positions, k of which are empty
	function with k empty parameter positions	*function* with n(≥k) parameter positions, k of which are empty
	procedure with k empty parameter positions	(same)
	parameter occurring in a *procedure* (added as a primitive to the language).[6]	every string of symbols s, which does not contain the symbol "," (comma)
output parameters	single identifier (formal variable)	simple or subscripted variable
	array (as above for input parameters)	array (as above for input parameters)
	(formal) label	label

If a parameter is at the same time an input and output parameter this parameter must obviously meet the requirements of both input and output parameters.

Within a program, a procedure statement causes execution of the procedure called by the statement. The execution, however, is effected as though all formal parameters listed in the procedure declaration heading were replaced, throughout the procedure, by the actual parameters listed, in the corresponding position in the procedure statement.

This replacement may be considered to be a replacement of every occurrence within the procedure of the symbols (or sets of symbols) listed as formal parameters, by the symbols (or sets of symbols) listed as actual parameters in the corresponding positions of the procedure statement, after enclosing in parentheses every expression not enclosed completely in parentheses already. Furthermore, any *return* statement is to be replaced by a *go to* statement referring, by its label, to the statement following the *procedure* statement, which, if originally unlabeled, is treated as having been assigned a (unique) label during the replacement process.

The values assigned to, or computable by, the actual input parameters must be compatible with *type* declarations concerning the corresponding formal parameters which appear in the procedure.

For actual output parameters, only *type* declarations duplicating given *type* declarations for the corresponding formal parameters may be made.

Array declarations concerning actual parameters must duplicate, in corresponding subscript positions, array declarations referring to the corresponding formal parameters.

E. *DECLARATIONS* Δ

Declarations serve to state certain facts about entities referred to within the program. They have

[6] Within a program certain procedures may be called which are themselves not defined by procedure declarations in the program; e.g., input-output procedures. These procedures may require as parameters quantities *outside* the language; e.g., a string of characters providing input-output format information.

no operational meaning and within a given program their order of appearance is immaterial. They pertain to the entire program (or procedure) in which they occur and their affect is not alterable by the running history of the program.

1. *Type declarations*

   Type declarations serve to declare certain variables, or functions, to represent quantities of a given class, such as the class of integers or class of Boolean values.

   Form: $\Delta \sim$ *type* (I, I, ⌇⌇⌇, I, I[ ], ⌇⌇⌇, I[,], ⌇⌇⌇, I[,,], ⌇⌇⌇) where *type* is a symbolic representative of some *type* declarator such as *integer* or *boolean*, and the I are identifiers. Throughout the program, the variables, or functions named by the identifiers I, are constrained to refer only to quantities of the type indicated by the declarator. On the other hand, all variables, or functions which are to represent other than arbitrary real numbers must be so declared.

2. *Array declarations*

   Array declarations give the dimensions of multidimensional arrays of quantities.

   Form: $\Delta \sim$ *array* (I, I, ⌇⌇⌇, I[C : C'], I, I, ⌇⌇⌇, I[C : C'], ⌇⌇⌇) where *array* is the *array* declarator, the I are identifiers, and the "C" and "C' " are lists of integers separated by commas. Within each pair of brackets, the number of positions of C must be the same as the number of positions of C'.

   Each pair of lists enclosed in brackets [C : C'] indicates that the identifiers contained in the list I, I, ⌇⌇⌇, I immediately preceding it are the names of arrays with the following common properties:

   a. The number of positions of C is the number of dimensions of every array.
   b. The values of C and C' are the lower and upper bounds of values of the corresponding subscripts of every array.

   An array is defined only when all upper subscript bounds are not smaller than the corresponding lower bounds.

3. *Switch declarations*

   A *switch* declaration specifies the set of designational expressions represented by a *switch* variable. If used in a *go to* statement, its value specifies the label of the statement called by the *go to* statement (cf. *go to* statement).

   Form: $\Delta \sim$ *switch* $I := (D_1, D_2, \text{⌇⌇⌇}, D_n)$ where *switch* is the *switch* declarator, I is an identifier, and the D_i are designational expressions (cf. *go to* statement).

   The switch declaration declares the list D_1, D_2, ⌇⌇⌇, D_n to be a symbolic vector (the "*switch*"), the designational expression D_k being the k-th component. Reference is made to the *switch* by the switch variable I[E], where I is the *switch* identifier and E is a subscript expression. The *switch* variable, when used in *go to* statements, selects by the actual value of its subscript that component of the *switch* determining the label called for by the *go to* statement. A *switch* variable, being a designational expression, may appear as a component of a switch.

4. *Function declarations*

   A *function* declaration declares a given expression to be a function of certain of its variables. Thereby, the declaration gives (for certain simple functions) the computing rule for assigning values to the function (cf. *functions*) whenever this function appears in an expression.

   Form: $\Delta \sim I_n$ (I, I, ⌇⌇⌇, I) :=E where the I are identifiers and E is an expression which, among its constituents, may contain simple variables named by identifiers appearing in the parentheses.

   The identifier I_n is the function name. The identifiers in parentheses designate the formal parameters of the function.

   Whenever the function I_n (P, P, ⌇⌇⌇, P) appears in an expression (a *function call*) the value

assigned to the function in actual computation is the computed value of the defining expression E. For the evaluation, every variable V which is listed as a parameter I in the *function declaration* is assigned the current value of the actual parameter P in the corresponding position of the parameter list of the function in the *function call.* The (formal) variables V in E which are listed as parameters in the declaration bear no relationship to variables possessing the same identifier, but appearing elsewhere in the program. All variables other than parameters appearing in E have values as currently assigned in the program.

Example:

$$I(Z) := Z+3 \times y$$

. . . . . .

$$\text{alpha} := q+I(h+9 \times mu)$$

In the statement assigning a value to alpha the computation is:

$$\text{alpha} := q+((h+9 \times mu)+3 \times y)$$

5. *Comment declarations*

Comment declarations are used to add to a program informal comments, possibly in a natural language, which have no meaning whatsoever in the algorithmic language, no effect on the program, and are intended only as additional information for the reader.

Form: $\Delta \sim$ comment S_j where *comment* is the comment declarator, and S_j is any string of symbols not containing the symbol ";".

6. *Procedure declarations*

A *procedure* declaration declares a program to be a closed unit (a procedure) which may be regarded as a single compound operation (in the sense of a generalized function) depending on a certain fixed set of input parameters, yielding a fixed set of results designated by output parameters, and having a fixed set of possible exits defining possible successors.

Execution of the procedure operation is initiated by a *procedure statement* which furnishes values for the input parameters, assigns the results to certain variables as output parameters, and assigns labels to the exits.

Form: $\Delta \sim$ *procedure* I $(P_i) =: (P_o)$, $I(P_i) =: (P_o)$, ⁓⁓⁓, $I(P_i) =: (P_o)$

Δ; Δ; ⁓⁓⁓; Δ *begin* Σ; Σ; ⁓⁓⁓; Δ; Δ; ⁓⁓⁓; Σ; Σ *end*

Here the I are identifiers giving the names of the different procedures contained in the *procedure declaration.* Each P_i represents an ordered list of formal input parameters, each P_o a list of formal output parameters which includes any exits required by the corresponding procedures.

Some of the strings "$=: (P_o)$" defining outputs and exits may be missing, in which case the corresponding symbols "$I(P_i)$" define a procedure that may be called within expressions.

The Δs in front of the delimiter *begin* are declarations concerning only input and output parameters. The entire string of symbols from the declarator *procedure* (inclusive) up to the delimiter *begin* (exclusive) is the *procedure heading.*

Among the statements enclosed by the parentheses *begin* and *end* there must be, for each identifier I listed in the heading as a procedure name, exactly one statement labeled with this identifier, which then serves as the entry to the procedure. For each single output procedure $I(P_i)$ listed in the heading, a value must be assigned within the procedure by an assignment statement "I := E" where I is the identifier naming that procedure.

To each procedure listed in the heading, at least one *return* statement must correspond within the procedure. Some of these *return* statements may, however, be identical for different procedures listed in the heading.

Since a procedure is a self-contained (except for parameters) program, the defining rules for statements and declarations within procedures are those already given. A formal input parameter may be

a. a single identifier I (formal variable)

b. an *array* I [, , ⁓⁓⁓,] with k (k = 1, 2, . . .) empty subscript positions

c. a *Function* f (, , ⁓⁓⁓,) with k (k = 1, 2, . . .) empty parameter positions
d. a *Procedure* P (, , ⁓⁓⁓,) with k (k = 1, 2, . . .) empty parameter positions
e. an identifier occurring in a procedure which is added as a primitive to the language.

A formal output parameter may be
a. a single identifier (formal variable)
b. an *array* with k (k = 1, 2, . . .) empty subscript positions.

A formal (exit) label may be only a label.

A label is an admissible formal exit label if, within the procedure, it appears in *go to* statements or *switch* declarations.

An *array* declaration contained in the heading of the *procedure* declaration, and referring to a formal parameter, may contain expressions in its lists defining subscript ranges. These expressions may contain
a. number
b. formal input variables, arrays, and functions.

All identifiers and all labels contained in the procedure have identity only within the procedure, and have no relationship to identical identifiers or labels outside the procedure, with the exception of the labels identical to the different procedure names contained in the heading.

A procedure declaration, once made, is permanent, and the only identifiable constituents of the declaration are the procedure declaration heading, and the entrance labels. All rules of operations and declarations contained within the procedure may be considered to be in a language different from the algorithmic language. For this reason, a procedure may even be composed initially of statements given in a language other than the algorithmic language; e.g., a machine may be required for expressing input-output procedures.

A tagging system may be required to identify the language form in which procedures are expressed. The specific nature of such a system is not in the scope of this report.

Thus by using procedure declarations, new primitive elements may be added to the algorithmic language at will.

PART IIIa—BASIC SYMBOLS (α)

Delimiters (δ)

Operators	(ω)	$\sim$ + − × / = ≠ > ≥ ≤ > ¬ ∨ ∧ ≡
		$\sim$ *go to* *do* *return* *stop* *for* *if* *or if* *if either*
Separators	(σ)	$\sim$, ; . : := =: → $_{10}$
Brackets	(β)	$\sim$ ( ) [ ] ↑ ↓ *begin* *end*
Declarators	(ϕ)	$\sim$ *procedure* *switch* *array* *comment* *type**

Non-delimiters (μ)

Letters	(λ)	$\sim$ A ⁓⁓⁓ Z, a ⁓⁓⁓ z
Digits	(ζ)	$\sim$ 0 ⁓⁓⁓ 9

PART IIIb—SYNTACTIC SKELETON

Syllables

List	(C)	$\sim$ E, E, ⁓⁓⁓, E
Simple variable	(V)	$\sim$ I
Subscripted variable	(V)	$\sim$ I [E, E, ⁓⁓⁓, E]
Function	(F)	$\sim$ I (R)**
Statement label	(L)	$\sim$ I
Expression	(E)	$\sim$ G (See the appropriate sections in Part II for
Boolean expression	(B)	the composition rules)

* Representant.
** Where R $\sim$ P, P, P, ⁓⁓⁓, P, P

Designational expression (D) ~ L ~ I [E]
Parameters (P) ~ (See the appropriate sections in Part II for the composition rules)
Identifier (I) ~ λ μ μ μ ⋯ μ μ
Integer (G) ~ ζ ζ ζ ⋯ ζ
Number (N) ~ ζ ζ ζ ⋯ ζ . ζ ζ ⋯ ζ $_{10}$ ± G (G: may be empty)
Symbol string (S_δ) ~ α α α α ⋯ α α where α is not the particular δ given in the subscript

Statements (Σ)

Assignment statement (Σ) ~ V := E ~ V:= B
Compound statement (Σ) ~ *begin* Σ; Σ; ⋯; Σ *end* (at least one Σ)
Labelled statement (Σ) ~ L : Σ where Σ is unlabeled
Go to statement (Σ) ~ *go to* D
Do statement (Σ) ~ *do* L_1, L_2 (S→I, S→I, ⋯, S→I) (first: may be empty; second: may be empty)
Quantifier statement (Σ) ~ *if* B
(Σ) ~ *for* V := C
(Σ) ~ *for* V := E (E) E, E (E) E, ⋯, E (E) E
Alternative (Σ) ~ *if either* B_1; Σ_1; *or if* B_2; Σ_2; ⋯; *or if* B_k; Σ_k *end*
Stop and return statements (Σ) ~ *stop* ~ *return*
Procedure call statement (Σ) ~ I (R) =: (R)

Declarations (Δ)

Function declaration (Δ) ~ I (R) := E
Switch declaration (Δ) ~ *switch* I := (D, D, ⋯, D)
Symbol classification declaration (Δ) ~ *type* (I, I, ⋯, I)
Comment declaration (Δ) ~ *comment* S;
Array declaration (Δ) ~ *array* (I, I, ⋯, I [C : C'], I, ⋯)
Procedure declaration (Δ)

(Δ) ~ procedure I (R) =: (R) I (R) =: (R) ⋯ I (R) =: (R)
(first: may be empty; second: may be empty; third: may be empty)
Δ; Δ; ⋯; Δ; *begin* Σ; Σ; ⋯ Δ; Δ; ⋯ Σ; Σ *end*

PART IIIc—PUBLICATION LANGUAGE

As stated in the introduction, the reference language is a link between hardware languages and hand-written, typed or printed documentation. For transliteration between the reference language and a language suitable for publications (for example, lectures in Numerical Analysis) the following transliteration rules may be used:

Reference Language		*Publication Language*
subscript brackets	[ ]	lowering of the line between the brackets
exponentiation parentheses	↑ ↓	raising of the line between the arrows
parentheses	( )	any form of parentheses, brackets, braces
basis of ten	10	raising of the ten and of the following integral number, inserting of the intended multiplication sign
statement separator	;	*line convention*: each statement on a separate line *may be used.*

Furthermore, if line convention is used, the following changes may be simultaneously used:

multiplication cross	×	multiplication dot	.
decimal point	.	decimal comma	,
separation mark	,	any common non-ambiguous separation mark	

Example: integration of a function F(x) by Simpson's Rule. The values of F(x) are supplied by an assumed existent function routine. The mesh size is halved until two successive Simpson sums agree within a prescribed error. During the mesh reduction F(x) is evaluated at most once for any x. A value V greater than the maximum absolute value attained by the function on the interval is required for initializing.

```
procedure     Simps (F( ), a, b, delta, V);
comment       a, b are the min and max, resp. of the points def. interval of integ. F( ) is the function to
              integrated.
              delta is the permissible difference between two successive Simpson sums V is greater than
              the maximum absolute value of F on a, b;
begin
Simps:        Ibar: =V×(b−a)
              n   : =1
              h   : =(b−a)/2
              J   : =h ×(F(a)+F(b) )
J1:           S   : =0;
  for         k   : =1 (1) n
              S   : =S+F (a+(2×k−1) ×h)
              I   : =J+4×h×S
  if          (delta < abs ( I−Ibar) ) (7)
begin         Ibar: =I
              J   := (I+J)/4
              n   :=2×n; h := h/2
              go to  Jl  end
              Simps := I/3
return
integer       (k, n)
  end         Simps
```

(7) abs (absolute value) is the name of a standard procedure always available to the programmer so that it need not be supplied as an input parameter.