Part 1. Introduction.

In 1955, as a result of the Darmstadt meeting on electronic computers, the GAMM (association for applied mathematics and mechanics in Germany), set up a commitee on programming (Programmierungsausschuß). Later a subcommitee 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 organisations and the Association for Computing Machinery (ACM) was held in Los Angeles on May 9 and 1o, 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 organisations 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 October 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 October 19, 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 commitee was then established by Dr. Carr, which represented computer users, computer manufacturers, and universities. This committee held three meetings starting on January 24, 1958 and discussed many technical details of programming language. The language that evolved from these meeting was oriented more toward problem Language than toward computer language and was based on several existing programming systems. On April 18, 1958 the committee appointed a sub-committee 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 ~~by~~ at the above-mentioned ACM-Ad-Hoc committee meeting of April 18, 1958) indicated many common features. Furthermore, the GAMM group 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. The ACM proposal was based on experience with several successful, working, problem oriented language.

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 May 27 to June 2, 1958 and attended by F.L.Bauer, H.Bottenbruch, H.Rutishauser and K. Samelson from the GAMM committee and by J.Backus, C.Katz, A.J.Perlis and J.H. Wegstein from 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.

---

1) In addition to the members of the conference, the following people participated in the preliminary work of these committees:

ACM (List of all who attended any of the three meetings)

GAMM P.Graeff, P.Läuchli, M.Paul, Dr.F.Penzlin

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 characters used 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.
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 tape language.
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 univoque 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 acceptea by a compiler for that compiler.

3. Each one of these must be accompanied by a special set of rules for transliterating from Publication language.

2. Summary of results

3. Acknowledgements

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

Part II. Description of the reference language

1) Structure of the language

As stated in the introduction, the algorithmic language will be developed here in terms of the reference language. This means that all objects defined within the language are represented by a given set of symbols. It is only in the use of symbols that the other languages mentioned in the introduction (publication languages, hardware language) may differ from the reference language, whereas structure, and content should be the same for all languages.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for description of calculating rules is the well known arithmetic expression containing the equally well known constituent numbers, variables, and functions. From expressions, the most simple, and most important, independent, and self contained units of the language are built up in the form of explicit formulae. They are called arithmetic statements.

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

Statements may be supported by declarations which give no operating rules, but inform about certain properties of objects appearing in statements, such as the class of numbers to be represented by a variable, the dimension of an array of numbers, or even the set of rules defining a function.

Sequences of statements, and declarations are called programs. However, whereas complete, and rigid formal rules for constructing 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 operational meaning of the sequence.

2) Basic Symbols

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

1) Letters λ (the alphabets of small, and of capital letters)
2) figures $\int$ (arabic numerals 0 ... 9)
3) delimiters δ consisting of

a) operators ω:

arithmetic operators	$+ - \times /$
relational operators	$< \leqslant = \geqslant > \neq$
logical operators	$\neg \vee \wedge \equiv$
sequential operators	go to, do, return, stop, for, if

b) separators σ: . , 10 ; : := =: →

c) brackets β : () [ ] ↑↓ begin end

d) declarators φ: procedure, array, switch, tree, type, comment

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

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 } 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 } are called
identifiers I

They have no inherent meaning, and serve for identifying purposes only.

3) Expressions.

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

a) (positive) Numbers N.

Form: $N \sim G \cdot G_{10} \pm G$

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 ± G. The following constituents of a number may be omitted:

The fractional part .oo...o of integer decimal numbers, the integer 1 in front of a scale factor, the + sign in the scale factor, the scale factor 1o ± o .

Examples:

4711

137.o6

$2.99997_{1o}{}^{1o}$

$1o^{-12}$

$3_{1o}{}^{-12}$

b_1) Simple Variables V

are designations for arbitray quantities (numbers) as in elementary arithmetics.

Form: V ~ I .

where I is an identifier as defined above.

Examples:

a

x11

ALPHA

b_2) Subscripted Variables V

designate quantities which are components of multidimensional arrays.

Form: V ~ I [l]

where l ~ 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. expressions).

Subscripts, however, are intrinsically integer valued, and whenever the value of a subscript expression is non integer, 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, complex, and so on, or Boolean. Finally, a special class of subscripted (logical) variables which like Boolean variables may assume only the two values „ true" and „ false", are the so called

tree-variables T ~ I [i].

These also are defined by special declaration. The subscripts of tree variables have the meaning of decimal classification, which describes a certain interdependence between all tree variables named by the same identifier, the "tree" (c.f. tree assignment statements).

If, for two such tree variables with different numbers of of subscripts, all existing subscripts are identical, the tree variable with the smaller number of subscripts is called a " predecessor" of the second one, which conversely, is a " successor" of the first one.

If, however , for two such tree variables with the same number of subscripts, all subscripts save the last are identical, the one with the lower value of this subscript is called a " lower neighbour", of the second one, which in turn is a " higher neighbour" of the first one.

c) Functions F

represent single numbers (function values), which result, by given sets of rules, from 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.

If the function is defined by a function declaration, admissible actual parameters are expressions compatible with the type of variables contained in the corresponding parameter positions of 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 may in general be chosen according to taste as in the case of variables. 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
entier	(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 mathematial notation.

d) Arithmetic expressions E

are defined as follows:

A number, a variable, which is not a Boolean, or tree variable, or a function is an expression

E ~ N
~ V
~ F.

Any expression E enclosed in (arithmetic) parentheses is an expression

E ~ (E).

Furthermore, expressions are defined recursively by the following composition rules, where E' is any expression the first symbol of which is neither " + " nor " - " ;

1. E ~ + E'
2. ~ - E'
3. ~ E+E'
4. ~ E-E'
5. ~ E×E'
6. ~ E/E'
7. ~ E↑E↓

The operators +, -, ×, / appearing in 1 through 6 have the conventional meaning.
The parentheses ↑ ↓ used in 7 denote exponentation, where the leading expression is the basis, 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 $(2^2)^n$

An arithmetic expression is a rule for computing one real number by executing the arithmetic operations indicated with 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 value assigned last (in the dynamic sense), and for functions F it is the value arising, with the actual values of the function parameters given in the expression, from the computing rules defining the function (cf. function declaration). x)

x) 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 first: × /
second: + -
applies.

x) Obviously, the value of an expression is undefined whenever
a) some constituents have not been assigned a value
b) a constituent which is a divisor has been assigned the value o.
However, no formal rules are given to detect these cases.

However, in order to avoid misunderstandings, redundant parentheses should be used to express for example $\frac{a\,b}{c}$ 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
Alpha
Degree
A[1,1]
A[j+k-2,j-k]
A[mu[s]]
a×sin(omega×t)
0.5×a[N×(N-1)/2, 0]
```

e) Boolean expressions B

are defined in a way analogous to arithmetic expressions:

B ~	0	(the truth value " false")
~	1	(the truth value " right")
~	V	(V being a Boolean variable by declaration)
~	T	(T being a tree variable by declaration)
~	(B)	
~	(E<E)	
~	(E≤E)	
~	(E=E)	
~	(E≠E)	
~	(E≥E)	
~	(E>E)	

Furthermore, Boolean expressions are defined recursively by the following composition rules

1.	B ~ ¬B	„not"
2.	~ B ∨ B	„ or"
3.	~ B ∧ B	„ and"
4.	~ B ≡ B	„ equivalent"

Boolean expressions, like arithmetic expressions, are rules for computing one truth value from the truth values of its constituents, by means of the standard operations of Boolean algebra (propositional calculus).

Interpretation will be from left to right, and precedence must be indicated by the use of parentheses.

Examples:

(x=0)

(x>0)∨(y<0)

(p∧¬q)∨(x≠y)

4) Statements Σ

Closed and selfcontained rules of operations are called Statements Σ. They are defined recursively in the following way.

1) Basic statements Σ are those described in this paragraph.

2) Strings of one or more statements[x] may be combined into a single (compound) statement by enclosing them into the "statement parentheses" begin and end. Single statements are separated by the statement separator (;).

begin Σ; Σ; ... ; Σ end

3) A statement may be made identifiable by attaching 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 constitute a statement called "labeled statement".

Σ ~ L : Σ

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 show distinctly the range of the compound statement.

Σ L : begin Σ ; Σ ; ... ; Σ end L;

a) Assignment statements

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

Form a): Σ ~ 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 either a Boolean, or a tree variable.

---

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

Generally, the arithmetic type of the variable V 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, and serves only to eliminate errors in value resulting from computational limitations.x)

Form b) $E \sim V := B$

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

Tree assignment statements.

A Boolean expression B may be used to assign a value to a tree variable T. The value actually assigned, however, is not just the value of B, but the value of

$$B \wedge \neg B' \wedge \neg B'' \ldots \wedge \neg B^{(n)} \wedge \bar{B} \wedge \bar{\bar{B}} \wedge \ldots \wedge \bar{\bar{\ldots}}\bar{B}$$

where B', B'',...,and $\bar{B}$, $\bar{\bar{B}}$,..., respectively, are the Boolean expressions used to define the " lower neighbours", and the "predecessors" of T respectively.

Furthermore, the following rules apply to the other components of the tree:

---

x) If, for example, i and k are integers, by declaration, and

$$m := i*k$$

m may be declared to be an integer, in which case the product will be rounded to the nearest integer if necessary.

On the other hand, if i, k and m are declared to be integers,

$$m := i/k$$

is contradictory, and therefore inadmissible.

In this case, the entier - function must be used, and if the nearest integer is wanted, the correct statement reads

$$m := ent(i/k + 0{,}5).$$

1) If the value assigned, according to the rule stated above, to a particular tree variable is the value " true", all " neighbours" of this tree variable, and all " successors" of these neighbours, are given the value " false".
2) If the value assigned is " false", all " successors", of the tree variable concerned are given the value " false" too.

The logical interdependence of tree variables within a tree is therefore isomorphic to decimal classifications, where each tree variable describes a case which is a subcase of its predecessors, whereas all neighbours are mutually exclusive parallel cases.

The use of the tree variables therefore is equivalent to a decimal classification of mutually exclusive cases and subcases which is quite conventional, and can be understood without recourse to formal assignment rules.

b) Go to statements

Normally, 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

Σ ~ go to D

The designational expression D specifies the label of the statement with which operation is to be resumed. It is a label L in which case L is the label specified by D, or a switch variable I[E] (cf. switch declarations) where I is an identifier and E a subscript expression.

In this case, the numerical value of E (or the integer nearest to it, if necessary), gives the number of an element of the switch named I { } by declaration. This element which is again a designational expression specifies the label to be used in the go to statement. If there is no element corresponding to the number given by E (this being too small or too large), no label is specified

by the designational expression, and the go to statemant is void. This label determination is obviously a recursive process, since the elements of the switch may again be switch variables.

Examples:

go to hell

go to exit [(i↑2↓- y↑2↓+1)/2]

where exit refers to

switch exit := [$D_1, D_2, \ldots, D_m$]

c) 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 Σ ~ 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 by-passed, and operations will be resumed with the next following statement.

Examples:

if (a>0); c:=a↑2↓+ b↑2↓ ;

if (a<0); c:=-a↑2↓+ b↑2↓ ÷

if (a=0); go to bed

d) For statements

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

Form

a) for V := l

b) for V := $E_i(E_s)E_e, \ldots, E_i(E_s)E_e$

where l is a list of expressions, and E_i, E_s, E_e are expressions, none of which may contain the variable V.

In form a) the value of each expression of the list (expressions taken in the order of listing) is assigned to the variable V, and the statement following the for statement is executed immediately after this assignment.

In form b), each group of expressions $E_i(E_s)E_e$ determines an arithmetic progression. The value of E_i is the initial value, E_s gives the value of the increment (step), and E_e determines the end value which is the last term of the progression contained in the closed interval $[E_i,E_e]$. However, a progression is empty, and specifies no value, if the half open intervals $(E_i,E_i+E_s]$ and $(E_i,E_e]$ do not intersect, that is if the intervall $(E_i,E_e]$ $\not\ni$ and the terms of the prograssion lie on opposite halves of the real line.

Each value of every profession (these again taken in the order of listing from left to right) is assigned to the variable V, and the statement following the for statement is executed immediately afterwards.

After execution of the statement following the for statement for the last value specified by the for statement, the next following statement is executed.

As for as the variable V is concerned, the for statement is an assignment statement, This means that V always has the value last assigned to it by the for statement even if complete execution of the for statement is prevented by a constituent of the statement following it (which may contain conditional go to statements).

Examples:

```
for I := 1(1)N; p:= p*x + A[I] ;
for a:= 1,3.5, 9.76 , ... , -13.75;
begin ---------
      --------- end
```

Since the values of V ate completely determined by the for statement, the following statement may not contain V on the left hand side of an assignment statement. Similarly, a jump (by means of a go to statement) to a constituent of this statement by passing the preceding for statement generally is inadmissible, since this leaves unspecified the sequence of values to be assigned to V.

e) 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 at the same time permits the substitution of certain constituents of the statement reused.

Form: Σ ~Do L_1, L_2 ($S_{\rightarrow}$ I, $S_{\rightarrow}$ I, ..., $S_{\rightarrow}$ I)

where L_1 and L_2 are labels, the $S_{\rightarrow}$ are strings of symbols not containing the separator (->) 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, all identifiers or labels contained in these statements which are listed on the right hand side of a separator (->) in the substitution list of the do statement, are replaced by the corresponding strings of symbols S on the left hand side of the separators (->). These strings S may be chosen freely with the one restriction that the substitutions produce formally correct statements.

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 copy of statements made on account of a do statement never contains a do statement.

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 or end delimiter of a compound statement lies in the range of the do, than the entire compound statement should be contained in this range. If this rule is not complied with the result of the do statement may not be the one desired.

f) Stop statements

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

Form: Σ~stop

g) Return statements

Return is a delimiter which indicates an operational end of a procedure. It may appear only in a procedure declaration, where no operational successor is given (cf. procedure statements)

Form: Σ~return

h) Procedure statements

A procedure statement serves to initiate execution of a procedure, that is a closed, selfcontained process with a fixed ordered set of input and output parameters, permanently defined by a procedure declaration (cf. procedure declaration).

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

Here I is an identifier which is the name of some procedure i.e. it appears in the heading of some procedure declaration.(cf. procedure declarations).

$P_i, P_i, \ldots P_i$ is the ordered list of actual input parameters specifying the input quantities to be processed by the procedure.

The list of actual output parameters $P_o, P_o, \ldots P_o$, specifies the variables, to which the results of the procedure will be assigned.

The list of actual exits $L, L, \ldots L$ gives the labels of statements with which operations may be resumed after execution of the procedure (which may contain different exits depending on intermediate results).

The procedure declaration defining the procedure called contains, in its heading, a string of symbols identical in form to the procedure statement. The formal parameters there in occupying input, and output parameter positions give complete information about admissible actual parameters by the following replacement rules:

	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	procedure with k empty parameter positions
	identifier occurring in a procedure which is added as a primitive to language	every string of symbols s, which does not contain the symbol "," (comma)
output parameters	single identifier (formal variable) array (as above, input)	simple or subscripted variable array (as above, input)
exits	(formal) label	label

If a parameter is at the same time input and output parameter, which may be necessary in the case of some procedures, this parameter must obviously meet requirements of both input and output parameters.

Within a program, a procedure statement causes execution of the procedure called by the statement, and given by the corresponding procedure declaration. Initially, however, all formal parameters listed in the procedure declaration heading are 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 occurence 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, all return statements are to be replaced by go to statements referring, by its label, to the statement following the procedure statement, which, if originally unlabeled, is treated as having been assigned an intermediary label by the replacement process.

The values assignable to, or computable by the actual input parameters must be compatible with type declarations concerning the corresponding formal parameters in the procedure declaration heading.

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, after replacement of formal parameters by actual parameters.

5) Declarations

Declarations serve to state, once and for all within a given program containing them, certain facts about entities referred to within the program. They have no operational meaning.

a) Type declarations

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

Form: $\Delta \sim$ type (I,I,...,I)

where type is a symbolic representative of some type declarator such as integer, complex, boolean, or tree, and the I are identifiers.

The variables, or functions named by the identifiers I are, throughout the program, constrained to refer only to quantities of the type indicated by the declarator, However, this constraint must be compatible with all values assignable (disregarding computational limitations) to these variables, or functions on account of statements of the program. (cf. arithmetical expressions for the case of integers). At the other hand, all variables, or numbers known to represent a given type of quantities only must be so declared. Variables, or functions not listed in a type declaration are automatically assumed to represent arbitary real numbers.

b) Array declarations

Array declarations give the dimensions of multi dimensional arrays of quantities.

Form: $\Delta \sim$ array (I,I,...,I[l:l'], I,I,...,I[l:l'],..)

where array is the array declarator, the I are identifiers, and the l, and l' are lists of expressions (separated by commas), each of which is of one of the following forms:

1. an integer

or, only if the array declaration is attached to a program declared to be a procedure

2. an expression containing only numbers, and formal variables which appear in the procedure declaration heading.

Within each pair of brackets, the numbers of positions of l must be the same as the number of positions of l'.

Each pair of lists enclosed in brackets [l:l'] 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 numbers of positions of l is the number of dimensions of every array.

b) the values represented by corresponding expressions of l and l' (which, in the case of expressions containing formal variables, are all possible values obtainable by assigning admissible values to these variables) are the lower and upper bounds of ~~results~~ values of the corresponding subscripts of every array. Only for subscript values with in the closed intervals given by these bounds, array elements are defined

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

c) Switch declarations △

Switch declarations specify the set of designational expressions represented by a switch variable which, in a go to statement, {Specifies the label of the statement} called by the go to statement (cf. go to statement)

Form: △ ~ switch I := (D_1,D_2,...,D_n)

where switch is the switch declarator, I is an identifier, and the D are designational expressions (cf. go to statements).

The list D_1, D_2, ..., D_n is, by the switch declaration, declared to be a symbolic vector (the "switch") the designational expression

D_k being the k th component.

Reference may be made to the switch by the switch variable I[E] where I is the identifier following the switch delimiter, and E is a subscript expression. The switch variable may be used only in go to statements, and specifies, by the actual value of its subscript, the component of the switch determining the label called for by the go to statement.

d) Function declarations Δ

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 parantheses designate the formal parameters of the function.

Whenever the function $I_N(P,P,...,P)$ appears in an expression, the value assigned to the function in actual computation is the value of the expression E. For evaluation, every variable V which is listed as a parameter I in the declaration, is assigned the actual value of the actual parameter P in the corresponding position of the parameter list of the function. The (formal) variables V in E which are listed as parameters bear no relationship to variables named by the same identifiers, but appearing otherwhere in the program. To all other variables appearing in E, and not listed as parameters, values must be assigned in the program.

e) Procedure declarations Δ

A procedure declaration declares a program to be a closed unit (a procedure) which may be regarded to be 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 (which may depend on intermediate results) leading to different 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 of existing statements to the exits.

Form:

$\Delta \sim$ procedure I (P_i) =: $(P_o:L)$, I(P_i) =: $(P_o:L)$,..,I(P_i) =: $(P_o:L)$;
$\Delta;\Delta;\ldots;\Delta;$
begin $\Sigma;\Sigma;\ldots;\Sigma;$ 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 and each L a list of formal labels, the exits of the corresponding procedures.

Some of the strings " =: $(P_o:L)$" defining outputs and exits may be missing in which case the corresponding symbols "I(P_i)" define a function (which is a special procedure).

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 by this identifier as a label.

For each function $I(P_i)$ listed in the heading, a value must be assigned within the procedure by an assignment statement "I := E", where I is the identifier giving the name of the function.

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.

For any do statement occurring within a procedure, the labels giving the range of the do statements must be labels of statements contained in the procedure, i.e. the range of the do must be completely contained in the procedure. No do statement may call for statements given outside the procedure.

A formal input parameter may be

a single identifier I (formal variable),

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

a function f (, ,...,) with k(k=1,2,...) empty parameter positions,

a procedure P (, ,...,) with k (k=1,2,...) empty parameter positions,

an identifier occurring in a procedure which is added as a primitive to the language.

A formal output parameter may be

a single identifier (formal variable)

an array with k (k=1,2,...) empty subscript positions.

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

An identifier or array is an admissible formal input parameter only if the corresponding simple or subscripted variable within the program appears only in expressions. No value may be assigned to it by an assignment statement, unless the identifier or array in question is also listed as a formal output parameter,

which must be assigned a value by an assignment statement at least once.

A label is an admissible formal exit label if, within the procedure, it appears only in go to statements or switch declaration. It must not be the label of a statement within the procedure.

Functions are admissible, formal input parameters only, if not covered by a function declaration, contained in the procedure.

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

1. numbers,
2. formal input variables, arrays, and functions.

All identifiers and labels contained in the procedure are identifiable 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. Each of these "entrance labels" indicates the initial statement (in the dynamic sense) of the corresponding procedure, with which operations begin upon a call by a corresponding procedure statement.

A procedure declaration, once made, is permanent, and the only constituents of the declaration identifiable from the outside 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 initially be composed of statements given in a language other than the algorithmic language, preferably a natural, or a machine language.

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

Comment declarations Δ

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

Form:

$\Delta \sim$ comment $S_;$

where comment is the comment declarator, and $S_;$ is any string of symbols not containing the symbol ";".

Part IIIa

1) Basic symbols

delimitors δ

operators ω

arithmetic	relational	logical	sequential
+	<	¬	go to
−	⩽	∨	do
×	=	∧	return
/	⩾	≡	stop
	>		for
	≠		if

separators σ

. 10 , ; : := =: ->

brackeets β

( )
[ ]
↑ ↓
begin end

declarators φ

procedure
switch
type x)
array
tree
comment

non - delimiters μ

letters

$\lambda \sim$ A through Z
a through z

figures

$\} \sim$ o through 9

---

x) Representant

Part III b

Syntactic skeleton

2) Basic constituents

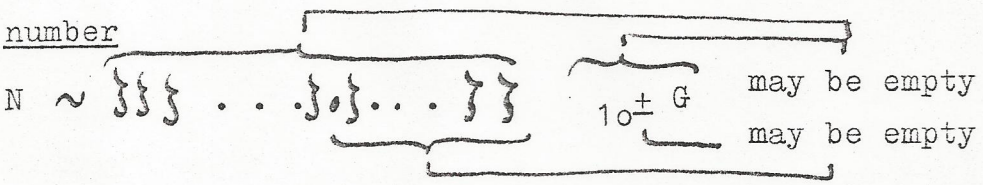
integer

G ~ }}} . . . }

identifier

I ~ λ μ μ μ . . . μ

number



list

ℓ ~ E , E , . . . , E

simple variable

V ~ I

subscripted variable

V ~ I [E , E , . . . , E]

tree variable

T ~ I [E , E , . . . , E]

function

F ~ I (P , P , . . . , P)

expression and Boolean expression

E
B For the composition rules see the appropriate sections in Part II

label

L ~ I
L ~ G

designational expression

D ~ L
D ~ I [E]

parameters

P For the composition rules see the appropriate sections in Part II

string of symbols

$S_\delta \sim \chi\chi \ldots \chi\chi$ where $\chi \neq \delta$, δ is a particular delimiter

3) Statements Σ

compound statement

$\Sigma \sim$ begin $\underbrace{\Sigma\ ;\ \Sigma\ ;\ \ldots\ ;\ \Sigma}_{\text{at laast one } \Sigma}$ end

labeled statement

$\Sigma \sim L : \Sigma$

assignment statement

$\Sigma \sim V := E$
$\sim V := B$

tree assignment statement

$\Sigma \sim T := B$

go to-statement

$\Sigma \sim$ go to D

IF-statement

$\Sigma \sim$ if B

for-statement

$\Sigma \sim$ for $V := \ell$
$\sim$ for $V := E\ (\ E\ )\ E\ ,\ E\ (\ E\ )\ E\ ,\ \ldots\ ,\ E\ (\ E\ )\ E$

do-statement

$\Sigma \sim$ do $\underbrace{L\ ,\ L}_{\text{may be empty}}\ (\ \underbrace{S_{\rightarrow} \rightarrow I\ ,\ S_{\rightarrow} \rightarrow I,\ \ldots\ ,\ S_{\rightarrow} \rightarrow I}_{\text{may be empty}}\ )$

stop- and return-statement

$\Sigma \sim$ stop
$\sim$ return

procedure statement

$\Sigma \sim I\ (\ R\ ) =: (\ R\ )$ where $R \sim P\ ,\ P\ ,\ \ldots\ ,\ P$

4) Declarations Δ

type declaration

$\Delta \sim$ type (I , I , ..., I)

array declaration

$\Delta \sim$ array (I , I , . . . , I $[\ell:\ell]$, I , . . . , I $[\ell:\ell]$, I ,. . .)

switch declaration

$\Delta \sim$ switch I := [D , D , . . . , D]

function declaration

$\Delta \sim$ I (R) := E

procedure declaration

$\Delta \sim$ procedure I (R) $\underbrace{=: (R)}_{\text{may be empty}}$, I (R) $\underbrace{=: (R)}_{\text{may be empty}}$, ... I (R) $\underbrace{=: (R)}_{\text{may be empty}}$;

$$\underbrace{\Delta ; \Delta ; \ldots ; \Delta}_{\text{may be empty}}$$

begin $\underbrace{\Sigma ; \Sigma ; . . . ; \Sigma}_{\text{at least one } \Sigma}$ end

where R $\sim$ (P , P , P , . . . , P , P)

comment declaration

$\Delta \sim$ comment S ;

Part IIIc

Publication language

As stated in the introduction, the reference language is a link between hardware languages and handwritten, typed or printed documentation. For transliteration between the reference language and a language suitable for publications, and for the use in lectures on Numerical Analysis the following

transliterations rules

may be used

reference language		publication language
subscript brackets	[ ]	lowering of the line between the brackets
exponentation parentheses	↑ ↓	raising of the line between the parentheses
parentheses	( )	any form of parentheses, brackets, braces
basis of ten	10	raising of the ten and of the following integral number, inserting of the multiplication sign
statement separator	;	line convention: any statement in a new line

Furthermore, if line convention is obeyed, the following changes may be made simultaneously

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