C | The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference

By J. W. Backus, International Business Machines Corp., New York (USA)

This paper gives a summary of the syntax and interpretation rules of the proposed international algebraic language put forward by the Zurich ACM-GAMM Conference, followed by a formal, complete presentation of the same information. Notations are presented for numbers, numerical variables, Boolean variables, relations, n-dimensional arrays, functions, operators and algebraic expressions. Means are provided in the language for the assignment of values to variables, conditional execution of statements, iterative procedures, formation of compound statements from sequences of statements, definition of new statements for arbitrary procedures, and the re-use and alteration of program segments.
The proposed language is intended to provide convenient and concise means for expressing virtually all procedures of numerical computation while employing relatively few syntactical rules and types of statement.

La syntaxe et la sémantique de langage algébraic international proposé par la Conférence de Zurich (*ACM et GAMM*). L'auteur caractérise brièvement la syntaxe et les règles d'interprétation du langage algébrique international proposé à la Conférence de Zurich (ACM-GAMM) puis en donne un exposé formel et complet. Il indique les notations utilisées pour désigner les nombres, les variables numériques ou booléennes, les relations, les agencements pluri-dimensionnels, les fonctions, les opérateurs et les expressions algébriques. Ce langage permet d'exprimer différentes opérations: affectation de valeurs aux variables, exécution conditionnelle des expressions, procédés itératifs, formation d'expressions complexes à partir d'une suite d'expressions élémentaires, définition de nouvelles expressions pour des opérations arbitraires, réemploi et modification de certaines parties du programme.
Le langage envisagé est conçu pour permettre d'exprimer la quasi totalité des procédés de calcul numérique de manière commode et concise à l'aide d'un nombre relativement restreint de règles de syntaxe et d'expressions-types.

Satz- und Wortbedeutungslehre der vorgeschlagenen Internationalen Algebraischen Sprache der Züricher ACM-GAMM-Konferenz. Der Bericht gibt eine erklärende Zusammenfassung der Satzlehre und Darstellungsregeln der vorgeschlagenen Internationalen Algebraischen Sprache, herausgebracht von der Züricher ACM-GAMM-Konferenz. Es folgt dann eine formale, vollständige Zusammenstellung desselben Stoffes. Symbole für Zahlen, numerische Variable, Boolesche Variable, Beziehungen, n-dimensionale Bereiche, Funktionen, Operatoren und algebraische Ausdrücke werden erläutert. In der Sprache sind Hilfsmittel vorgesehen, um Variablen Werte zuzuschreiben, bedingte Ausführung von Sätzen, Iterationsverfahren, die Bildung zusammengesetzter Sätze aus Folgen von Sätzen, Definition neuer Sätze für beliebige Verfahren, die Wiederverwendung und Änderung von Programmteilen ausführen zu können.
Die vorgeschlagene Sprache soll bequeme und kurze Ausdrücke haben, um wirklich alle Verfahren der numerischen Rechentechnik ausdrücken zu können. Dabei soll sie möglichst wenig Regeln der Satzlehre und Satzarten verwenden.

Синтаксис и сематика международного алгебраического языка, предложенного на Цюрихской конференции, связанной Ассоциацией по вычислительным машинам и Обществом прикладной математики и механики. В докладе кратко изложены правила синтаксиса и интерпретации международного алгебраического языка, предложенного на Цюрихской Конференции; затем дано формальное полное представление той же самой информации. Даны обозначения для численных, переменных. булевых переменных, зависимостей размерных множеств, функций, операторов и алгебраических выражений. В языке предусмотрены средства для задания значений переменных, условного выполнения утверждений, итеративных процессов, образования составных утверждений из последовательностей утверждений, опреде-

ления новых утверждений для произвольных процессов, повторного использования и изменения отрезков программы.
Предлагаемый язык предназначен для обеспечения удобных и точных средств выражения, в сущности, всех процессов цифрового вычисления, хотя он и требует относительно мало синтаксических правил и типов утверждений.

La sintaxis y la semántica del lenguaje algebraico internacional propuesto en la Conferencia ACM-GAMM de Zurich. Este artículo resume la sintaxis y las reglas de interpretación del lenguaje algebráico internacional propuesto en la Conferencia ACM-GAMM de Zurich, seguida de una presentación formal completa de la misma información. Las notaciones se refieren a números, variables numéricas, variables booleanas, relaciones, disposiciones n-dimensionales, funciones, operadores y expresiones algebráicas. Se proporcionan los medios adecuados en el lenguaje para las asignaciones de valores a las variables, la ejecución condicional de las proposiciones; los procedimientos iterativos, la formación de proposiciones compuestas a partir de secuencias de proposiciones, la definición de nuevas proposiciones por procedimientos arbitrarios y la reutilización y modificación de partes del programa.
Se considera que el lenguaje propuesto proporciona los instrumentos adecuados y suficientemente concisos para expresar virtualmente todos los procedimientos de cálculo numérico, aún empleando un número relativamente pequeño de reglas sintácticas y de tipos de proposición.

1. General

In May 1958 a conference in Zurich completed a joint project of the ACM (Association for Computing Machinery) and the GAMM (Association for Applied Mathematics and Mechanics) by approving a proposed International Algebraic Language (IAL) or ALGOL. Other papers to be presented here discuss the history of the project and various aspects of the language. It is the purpose of the present paper to describe precisely both the syntax and semantics of IAL. The description given here will be that of the so-called "reference language" which, for definiteness, employs a specific set of symbols and notational conventions.
It should be kept in mind, however, that the reference language form of IAL exists primarily for the purpose of describing the rules of construction of the language and its meanings. In actual use, a variety of symbolizations and notational conventions are envisaged, each representation being a fairly direct transliteration of the reference language and having the same rules of syntax and semantics. "Hardware representations" will generally use a smaller set of symbols than that employed in the reference language and will be suitable for mechanical translation into machine programs by a given type of machine. On the other hand, the "publication forms" of the language will employ many of the notational conventions of mathematics (e.g., exponents, subscripts, Greek letters) and will be used in human communication of IAL programs.
The description of the reference language is given in two parts. Chapter 2 gives a brief informal description of the major elements of the language in terms of examples. Chapter 3 gives a formal description of the structure of the language.

2. Informal description

2.1 *General*

An IAL program is a sequence of *statements* which may be interspersed with certain *declarations*. Each statement describes a rule of computation and, explicitly or implicitly, specifies a successor-statement (the successor being the next statement in the program unless otherwise stated). The computing rule given by a program is, of course, the sequence of computations specified by the statements when taken in the order provided by successor-relations. Declarations state certain facts which are to obtain throughout the computation. Their location in the program is generally unimportant.
The following paragraphs present some of the important properties of expressions and statements, with examples.

2.2 *Algebraic expressions*

a) Algebraic expressions are composed of variables, constants, functions, and operators for the usual arithmetic operations.
b) Names of variables, functions and other entities are composed of alphabetic and numeric characters, the first being alphabetic. A sequence of characters may be of any length.
c) It is understood that the arithmetic operations in an expression signify a floating point approximation of real number arithmetic. Some variables may be declared to be integer-valued. The effect of such a declaration is that any value which is to be assigned to such a variable is first rounded to the nearest integer.
d) Array declarations (cf. 2.5) may state that certain names are the names of arrays of data of various dimensions. These names may appear in expressions as *subscripted variables* which are followed by a sequence of *subscripts* enclosed in a pair of square brackets, the number of subscript corresponding to the dimensionality of the array and each subscript separated from the next by a comma. Since parentheses are used to enclose function arguments (and subexpressions), it is easy to distinguish between functions and subscripted variables. A subscript may be any expression; the value of the subscript is the integer formed by rounding the value of the expression (if it is not already an integer).

Examples of algebraic expressions:

1) $X \uparrow 2 \downarrow + Y \uparrow 2 \downarrow$
means: $X^2 + Y^2$

2) $3.14 \times (\text{alpha} + \sin(x))$
means: $3.14 \times (\alpha + \sin x)$

3) $(b/2 - \text{sqrt}(a \times c - (b/2) \uparrow 2 \downarrow)/a$
means: $(b/2 - \sqrt{a \times c - (b/2)^2}/a$

4) $R[i + 1, j] \times S[i, j + 1]$
means: $R_{i+1,j} \times S_{i,j+1}$.

In particular, if i and j both have the value 1 (or 1,2) when the above expression is encountered, it is then equivalent to: $R_{2,1} \times S_{1,2}$.

2.3 *Boolean expressions*

Some variables or arrays of quantities may be declared to be Boolean-valued (i.e., having only the value 0, "false", or 1, "true"). These variables, the constants 0 and 1, Boolean-valued functions, and subexpressions of the form:

$$(E\, rE')$$

may be combined with the Boolean operators "and", "or", "not", and "equivalent" to form a Boolean expression. (In the above E and E′ are algebraic expressions and r is a relation, e.g., (x/y > 1).)

Examples of Boolean expressions:

1. $(X > Y) \vee (Y > Z \uparrow 2 \downarrow)$

This Boolean expression is true if, and only if, either X is greater than Y or Y is greater than Z^2 or both. The operator is that for "or".

2. $((\neg A) \wedge B) \vee (A \wedge (X = Y + 1))$

Here A and B must be Boolean-valued variables. The expression is true when either A is false and B is true or when A is true and X equals Y + 1.

2.4 *Statements*

2.41 a) Statements may be either *basic* or *compound* (formed from a sequence of statements enclosed in the *statement parentheses*, **begin** and **end**). Statements are separated one from the next, by the separator ";".

Example of a compound statement:

begin a[i]: = b [i] + c [i]; x: = a [i] **end**

In general, if S_1, S_2, ···, S_n are statements then

begin S_1; S_2; ; ; S_n **end**

is a statement.

b) A statement may be labeled, as follows:

L:S

where L is a name or an integer; L thus becomes the label of statement S.

2.42 Assignment statements

Assignment statements direct that an expression is to be evaluated and assigned as the value of a variable.

Examples of assignment statements:

a) x: = a + b

means: assign the value of a + b as the value of x.

b) alpha: = 4.63

c) r [i, j]: = s [i] + t [j]

d) B: = (x > 0) ∧ A

here A and B are Boolean variables and B receives the truth-value of the Boolean expression on the right.

2.43 **go to** statements

a) A **go to** statement may specify some statement, other than the statement which follows it, as its successor in the computing process described by the program. This may be done explicitly by writing the label of the desired successor as follows:

go to A

where A is the label of some statement in the program. Or, the successor of the **go to** statement may be made to depend upon the value of some expression by the use of a *switch variable* whose subscript is the desired expression. The value of a switch variable is a label. Thus if "branch" is the name of a switch variable, then

go to branch [i + j]

has as its successor that statement whose label is the value of branch [i + j].

b) Switch variables are defined by *switch declarations* (2.53). Thus the switch variable "branch" might be defined by the following declaration:

switch branch (A1, A2, B1, B2)

where A1, A2, B1, B2 are labels of statements. The value of branch [i + j] is then the (i + j)th label in the sequence (or the nth label, where n is the integer formed by rounding i + j). For example, if i + j equals 3, then the value of branch [i + j] is B1, and in that instance, "**go to** branch [i + j]" has the same significance as "**go to** B1".

c) Switch variables may be used in the definition of other switch variables. Thus the following two switch declarations might appear in a program:

switch branch (A1, A2, B1, B2)

switch fork (branch [i], A1, B1).

In this case if i equals 4 and j equals 1, the value of fork [j] is B2.

d) A switch variable may have no value in some cases. For example, branch [i] as defined above has no value if the integer nearest i is less than 1 or greather than 4. In such an instance the successor of "**go to** branch [i]" is the next statement in the program.

2.44 **if** clauses and **if** statements:

The **if** clause, "**if** B", where B is some Boolean expression, combines with the statement, S, following it to form a compound statement. This compound statement has the same effect as S if B is true and has the effect of "no operation" if B is false.

Example of the use of **if** statements:

if (a > 0); x: = 1; **if** (a = 0); **begin** x: = y;
p: = 0 **end**;
if (a < 0); **go to** B; S

In the above program segment statement S is encountered unless a is negative, in which case "**go to** B" is executed; if a is positive and non-zero, x is 1 and p is unaffected when S is reached, otherwise x equals y and p equals 0.

2.45 **for** clauses and **for** statements:

a) A **for** clause, like an **if** clause, combines with the statement, S, following it to form a compound statement. A **for** statement specifies that a given variable take on a succession of values and that the governed statement, S, be executed once for each value of the variable. Thus,

for x: = 1, r + s, k, 6.3, —10; a: = (a + x)/x

causes the governed statement to be repeated five times, once for each of the listed values of x. Since the statement governed by a **for** clause may be an involved compound statement which may include other **for** statements, complex recursive procedures may be easily specified by their use.

b) Sequences of values may also be given as arithmetic progressions:

for i: = a ↑ 2 ↓ (b/2) y + 1

this clause directs that i should assume the values: a^2, $a^2 + b/2$, $a^2 + b$, ···, $a^2 + nb/2$, ··· until y + 1 is reached but not passed. Several such progressions may be included in a single **for** clause; e.g.

for r: = 1(3)10, 12(4)20, 21(—7)1

gives the sequence of r-values: 1, 4, 7, 10, 12, 16, 20, 21, 14, 7.

2.46 **converge** statements

a) A **converge** statement preceding a compound statement S causes those substatements comprising S which do not explicitly specify a successor, to have the statement following S as their common successor. However, in the case of an **if** statement, only the successor of the governed statement is altered. The successor of an **if** statement with a false Boolean expression is the next statement. Thus:

converge; **begin if** B_1; S_1; **if** B_2; S_2; **if** B_3; S_3 **end**; S_4

causes that single statement S_i to be executed which corresponds to the first true Boolean expression B_i; this S_i is then succeeded by S_4. Thus if B_1, B_2 and B_3 are all true, the effect is to execute S_1 and then S_4.

b) A **converge** statement may also be used to conveniently select a single statement for execution as follows:

converge; **begin go to** branch [i]; L_1; S_1; L_2 : S_2;
L_3 : S_3 **end**; S_4

If "branch [i]" takes on the values L_1, L_2, L_3 for i = 1, 2, 3, then the statement following **converge** has the effect of the single statement S_i followed by S_4.

2.47 **do** statement

a) A **do** statement is simply a shorthand way of rewriting a segment of a program which appears elsewhere. Thus,

do A; x: = y + z; A: r: = x + r

is another way of writing the following:

B: r: = x + r; x: = y + z; A: r: =x + r

where the copied statement has been given a new label, B, simply to avoid having two statements with the same label. The choice of new labels in the copy is immaterial as long as the reassignment is done consistently throughout the copy.

b) A **do** statement may specify a sequence of statements to be copied by giving two labels.

do R, S

thus represents the sequence of statements from the one labeled R through the one labeled S.

c) A **do** statement may require that certain alterations are to be made in the copy which it represents.

do R, S (x + y → x, a [i] → beta)

stands for the segment from R through S but with the name "x" replaced by "x + y" everywhere it appears and with "beta" replaced by "a [i]".

2.48 **stop** statements

stop statements have no successor, hence they signify the operational end of the process described by a program.

2.49 **return** statements

A return statement, written "**return**", is used only in programs which define the meaning of procedure statements. It signifies that the defined procedure has been completed and that the successor of the procedure statement which invoked the defining program is the next statement in the invoking program.

2.410 Procedure statements

A procedure statement indicates that some particular process is to be performed, employing those entities indicated by input parameters and producing results which are designated by the output parameters. (Some procedure statements may also indicate specific successor statements by a list of labels or switch variables which are associated with various exit conditions). Thus

integrate (F (r), a, b, e) = : (int) : (L)

might indicate that $\int_a^b F(r, x)\,dx$ is to be found with an error less than e (if possible with the given procedure), that the result is to be assigned as the value of "int" and that if the desired result is not obtained the successor of this statement should be the one labelled L.

Some procedure statements may simply be added to IAL as primitives (and realized by machine language subroutines). Others may refer, by the name which stands in front (e.g., "integrate"), to an IAL program (cf procedure declarations 2.56) which represents the process indicated.

Some procedure statements may not have any input parameters, others may have no output parameters. Thus a procedure statement which causes one number to be obtained from an external medium might be,

in = : (a)

indicating that the number should be assigned as the value of a. Another might be

print (r, s, t).

Another procedure statement may specify the replacement of a matrix, A, by its inverse:

invert (A [,] = : (A [,])

In general, it is intended that virtually any process can be expressed by a procedure statement.

2.5 *Declarations*

Declarations in an IAL program state certain facts which are to obtain throughout the program. They may appear at any point in the program, but their position in it is not significant.

2.51 Type declarations

A type declaration, **boolean** or **integer**, asserts that the value of a variable or function or element of an array will always be of the given type. Thus,

boolean (a, b, c)

asserts that any value assigned to entities having the names a, b, or c will be changed to a zero or a one according to some rule, t, (e.g., t (x) = 1 when x = 1, otherwise t (x) = 0). Thus

a : = x

in a program containing the above declaration would be equivalent to:

a : = t (x)

or again the appearance of a function b (x) in an expression in the program would be equivalent to t (b (x)).

Similarly,

integer (r, s)

constrains entities having names r or s to have integer values by rounding non-integer values to the nearest integer.

2.52 **array** declarations

array declarations specify the upper and lower bounds for the subscripts which may be meaningfully associated with an array. Thus

array (a, b, c [1 : 100], r, s [—10,1 : + 10,50])

indicates that a, b, c are one-dimensional arrays whose subscripts have meaning only in the range 1 to 100 (inclusive), and that r, s are two-dimensional arrays whose first subscript should lie between —10 and + 10 and the second between 1 and 50 (inclusive).

2.53 **switch** declarations

switch declarations have already been described under the heading, "**go to** statements" (2.43).

2.54 Function declarations

Certain functions may be defined for use within a program by a function declaration as follows:

g (x, y) : = w + x + f (y, x)

If, in the program containing the above declaration, the following appears: g (t/v, h (v)), then its value is given by the expression:

w + t/v + f (h (v), t/v)

Note, in this case, that g (x, y) is always a function of w even though it does not appear as an argument.

2.55 **comment** declarations

comment declarations do not affect the behavior of the program, they simply provide the ability to incorporate verbal comments in a program. For example:

comment This is the end of Section 1.

The comment may be arbitrarily long provided it does not contain a statement separator, semicolon.

2.56 Procedure declarations

Procedure declarations are unique among declarations: precisely one procedure declaration must precede the program it refers to. Such a declaration indicates that the IAL program following defines one or more procedure statements and/or functions. For example:

```
procedure branch (a): (L1, L2, L3);
begin branch: if (a < 0); go to L1; if (a = 0);
                  go to L2; if (a > 0); go to L3 end
```

This comprises a procedure declaration followed by a program of one compound statement. The procedure statement defined thereby is one named "branch" which selects one of three successors according as the value of the input is negative, zero, or positive. Thus the procedure statement,

```
branch (x — f(y)) : (a[i], BB, MM)
```

used in any program means, by virtue of its defining program: go to the statement indicated by the switch variable a[i] when x — f(y) is negative, if it is zero, go to BB, and if it is positive, go to MM.
One more example:

```
procedure root (a, b), ckroot (a, b) = : (c) : (d);
begin ckroot: if (a — b < 0); go to d;
    root : c : = sqrt (a — b); root : = c;
    return end
```

The procedure declaration, together with the program, defines a function, root, and a procedure statement, named "ckroot". The calculation of the function begins with the statement labelled root, and its value is that of the variable, root, when **return** is encountered. The program for ckroot begins at the first statement, labelled ckroot.
The process corresponding to

```
ckroot (x, y/z) = : (a) : (GL)
```

therefore either gives a the value sqrt (x — y/z) or, if x—y/z is negative, specifies the statement labelled GL as its successor. On the other hand,

```
v : = r + root (m, n)
```

results in v having the value r + sqrt (m — n) regardless of the sign of m — n.

3. Formal description

3.1 *General*

The Zurich ACM-GAMM Conference had two principal motives in proposing the IAL:

a) To provide a means of communicating numerical methods and other procedures between people.

and,

b) To provide a means for realizing a stated process on a variety of machines with the only differences between the realizations of a program on two machines being those imposed by differences in word lengths, overflow conditions and the like.

It appears that IAL, as given by informal descriptions here and elsewhere [1], suffices rather well for goal (a). However, if programs are to be written for a variety of machines to translate from IAL to the machine's language in such a way that goal (b) is satisfied, two requirements must be met:

c) There must exist a precise description of those sequences of symbols which constitute legal IAL programs. Otherwise it will often be the case that a program which is legal and translatable for one translating program will not be so with respect to another.

d) For every legal program there must be a precise description of its "meaning", the process or transformation which it describes, if any. Otherwise the machine language programs obtained by two translating programs from a single IAL program may behave differently in one or more crucial respects.

Heretofore there has existed no formal description of a machine-independent language (other than that provided implicitly by a complete translating program) which has met either of the two requirements above. Consequently, as anyone who has actually been involved in writing a translating program can testify, a large number of decisions affecting membership in the class of legal programs and the meaning of legal programs must be made during the construction of the translating program.
If, therefore, IAL is left in its present state of incomplete and informal description, it is likely that many man-years may be invested in producing a number of translating programs which will not reliably produce equivalent machine programs. In this case the great potential benefits and savings offered by goal (b) will be lost.
The author had hoped to complete a formal description of the set of legal IAL programs and of their meanings in time to present it here. Only the description of legal programs has been completed however. Therefore the formal treatment of the semantics of legal programs will be included in a subsequent paper. The work which has already been done in this direction indicates a need for minor modifications in IAL to facilitate its completion. These changes will require the approval of the appropriate groups. Since some of these changes are present in both chapters 2 and 3 of this paper, no official status for the descriptions contained in it should be assumed, although the changes are few and, for the most part, slight.

3.2 *Syntax of IAL*

In the description of IAL syntax which follows we shall need some metalinguistic conventions for characterizing various strings of symbols. To begin, we shall need *metalinguistic formulae*. Their interpretation is best explained by an example:

⟨ab⟩ : ≡ (*or* [*or* ⟨ab⟩ (*or* ⟨ab⟩ ⟨d⟩

Sequences of characters enclosed in "⟨⟩" represent metalinguistic variables whose values are strings of symbols. The marks ": ≡" and "*or*" are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the strings denoted. Thus the formula above gives a recursive rule for the formation of values of the variable ⟨ab⟩. It indicates that ⟨ab⟩ may have the value "(" or "[" or that given some legitimate value of ⟨ab⟩, another may be formed by following it with the character "(" or by following it with some value of the variable ⟨d⟩. If the values of ⟨d⟩ are the decimal digits, some values of ⟨ab⟩ are:

```
[(((1(37(
(12345(
(((
[86
```

3.31 Integers and numbers

⟨digit⟩ : ≡ 0 *or* 1 *or* 2 *or* 3 *or* 4 *or* 5 *or* 6 *or* 7 *or* 8 *or* 9

⟨integer⟩ : ≡ ⟨digit⟩ *or* ⟨integer⟩ ⟨digit⟩

⟨dn⟩ : ≡ ⟨integer⟩ *or* ⟨integer⟩ *or* ⟨integer⟩ *or* ⟨dn⟩ ⟨integer⟩

⟨si⟩ : ≡ + ⟨integer⟩ *or* — ⟨integer⟩ or ⟨integer⟩

⟨en⟩ : ≡ ⟨dn⟩ $_{10}$⟨si⟩ *or* $_{10}$⟨si⟩

⟨number⟩ : ≡ ⟨integer⟩ *or* ⟨dn⟩ *or* ⟨en⟩

3.32 Identifiers and variables

⟨letter⟩ : ≡ a *or* b *or* c *or* d *or* e *or* f *or* g *or* h *or* i *or* j *or* k *or* l *or* m *or* n *or* o *or* p *or* q *or* r *or* s *or* t *or* u *or* v *or* w *or* x *or* y *or* z *or* A *or* B *or* C *or* D *or* E *or* F *or* G *or* H *or* I *or* J *or* K *or* L *or* M *or* N *or* O *or* P *or* Q *or* R *or* S *or* T *or* U *or* V *or* W *or* X *or* Y *or* Z

⟨identifier⟩ : ≡ ⟨letter⟩ *or* ⟨identifier⟩ ⟨letter⟩ *or* ⟨identifier⟩ ⟨digit⟩

⟨id⟩ : ≡ ⟨identifier⟩

⟨variable⟩ : ≡ ⟨id⟩

3.33 Subscripted variables and arrays

⟨el⟩ : ≡ ⟨ar exp⟩ *or* ⟨el⟩, ⟨ar exp⟩

⟨subscr var⟩ : ≡ ⟨id⟩ [⟨el⟩]

⟨blank⟩ : ≡ ⟨the null string of characters⟩

⟨ppel⟩ : ≡ ⟨blank⟩ *or* ⟨ppel⟩ ⟨ar exp⟩

⟨pel⟩ : ≡ ⟨ppel⟩ *or* ⟨pel⟩, *or* ⟨pel⟩, ⟨ar exp⟩

⟨array⟩ : ≡ ⟨id⟩ [⟨pel⟩]

(see 3.35 for definition of ⟨ar exp⟩)

3.34 Parameters, functions and pure functions

⟨nq⟩ : ≡ ⟨any character other than a quote mark⟩

⟨arb⟩ : ≡ ⟨nq⟩ *or* ⟨arb⟩ ⟨nq⟩

⟨quoted string⟩ : ≡ "⟨arb⟩"

⟨param⟩ : ≡ ⟨exp⟩ *or* ⟨quoted string⟩ *or* ⟨array⟩ *or* ⟨pure function⟩ *or* ⟨pure procedure⟩

⟨param list⟩ : ≡ ⟨param⟩ *or* ⟨param list⟩, ⟨param⟩

⟨function⟩ : ≡ ⟨id⟩ (⟨param list⟩)

⟨pl⟩ : ≡ ⟨blank⟩ *or* ⟨pl⟩ ⟨param⟩

⟨ppl⟩ : ≡ ⟨pl⟩ *or* ⟨ppl⟩ *or* ⟨ppl⟩, ⟨param⟩

⟨pure function⟩ : ≡ ⟨id⟩ (⟨ppl⟩)

3.35 Arithmetic expressions, Boolean expressions, and expressions

⟨factor⟩ : ≡ ⟨number⟩ *or* ⟨function⟩ *or* ⟨variable⟩ *or* ⟨subscr var⟩ *or* (⟨ar exp⟩) *or* ⟨factor⟩ ↑ ⟨ar exp⟩ ↓

⟨term⟩ : ≡ ⟨factor⟩ *or* ⟨term⟩ × ⟨factor⟩ *or* ⟨term⟩ / ⟨factor⟩

⟨ar exp⟩ : ≡ ⟨term⟩ *or* + ⟨term⟩ *or* — ⟨term⟩ *or* ⟨ar exp⟩ + ⟨term⟩ *or* ⟨ar exp⟩ — ⟨term⟩

⟨ar exp A⟩ : ≡ ⟨ar exp⟩

⟨relation⟩ : ≡ < *or* > *or* ≤ *or* ≥ *or* = *or* ≠

⟨rel exp⟩ : ≡ (⟨ar exp⟩ ⟨relation⟩ ⟨ar exp A⟩)

⟨bool term⟩ : ≡ 0 *or* 1 *or* ⟨rel exp⟩ *or* ⟨function⟩ *or* ⟨variable⟩ *or* ⟨subscr var⟩ *or* (⟨bool exp⟩) *or* ¬ ⟨bool term⟩

⟨bool exp⟩ : ≡ ⟨bool term⟩ *or* ⟨bool exp⟩ ∨ ⟨bool term⟩ *or* ⟨bool exp⟩ ∧ ⟨bool term⟩ *or* ⟨bool exp⟩ ≡ ⟨bool term⟩

⟨exp⟩ : ≡ ⟨ar exp⟩ *or* ⟨bool exp⟩

3.41 Assignment statements

⟨left element⟩ : ≡ ⟨variable⟩ *or* ⟨subscr var⟩

⟨assnmt stmt⟩ : ≡ ⟨left element⟩ : = ⟨exp⟩

3.42 **go to** statements and designational expressions

⟨desig exp⟩ : ≡ ⟨id⟩ *or* ⟨integer⟩ *or* ⟨id⟩ [⟨exp⟩]

⟨**go to** stmt⟩ : ≡ **go to** ⟨desig exp⟩

3.43 **if** statements

⟨**if** clause⟩ : ≡ **if** ⟨bool exp⟩

⟨**if** stmt⟩ : ≡ ⟨**if** clause⟩; ⟨stmt⟩

3.44 **for** statements

⟨el⟩ : ≡ ⟨ar exp⟩ *or* ⟨el⟩, ⟨ar exp⟩

⟨unsigned exp⟩ : ≡ ⟨term⟩ *or* ⟨unsigned exp⟩ + ⟨term⟩ *or* ⟨unsigned exp⟩ — ⟨term⟩

⟨ap⟩ : ≡ ⟨ar exp⟩ (⟨ar exp⟩) ⟨unsigned exp⟩

⟨apl⟩ : ≡ ⟨ap⟩ *or* ⟨apl⟩, ⟨ap⟩

⟨**for** clause A⟩ : ≡ **for** ⟨left element⟩ : = ⟨el⟩

⟨**for** clause B⟩ : ≡ **for** ⟨left element⟩ : = ⟨apl⟩

⟨**for** stmt⟩ : ≡ ⟨**for** clause A⟩; ⟨stmt⟩ *or* ⟨**tor** clause B⟩; ⟨stmt⟩

3.45 **converge** statements

⟨**converge** clause⟩ : ≡ **converge**

⟨**converge** stmt⟩ : ≡ ⟨**converge** clause⟩; ⟨stmt⟩

3.46 **do** statements

⟨sc⟩ : ≡ ⟨any character other than "→"⟩

⟨ss⟩ : ≡ ⟨sc⟩ *or* ⟨ss⟩ ⟨sc⟩

⟨subsl⟩ : ≡ ⟨ss⟩ → ⟨id⟩ *or* ⟨subsl⟩, ⟨ss⟩ → ⟨id⟩

⟨label⟩ : ≡ ⟨id⟩ *or* ⟨integer⟩

⟨**do** stmt⟩ : ≡ **do** ⟨label⟩ *or* **do** ⟨label⟩ (⟨subsl⟩) *or* **do** ⟨label⟩, ⟨label⟩ *or* **do** ⟨label⟩, ⟨label⟩ (⟨subsl⟩)

3.47 **stop** statements

⟨**stop** stmt⟩ : ≡ **stop**

3.48 Procedure statements and pure procedures

⟨oe⟩ : ≡ ⟨left element⟩

⟨out list⟩ : ≡ ⟨oe⟩ *or* ⟨outlist⟩, ⟨oe⟩

⟨suc⟩ : ≡ ⟨label⟩ *or* ⟨id⟩ [⟨exp⟩]

⟨succr list⟩ : ≡ ⟨suc⟩ *or* ⟨succr list⟩, ⟨suc⟩

⟨A⟩ : ≡ = : (⟨out list⟩) *or* ⟨blank⟩

⟨B⟩ : ≡ : (⟨succr list⟩) *or* ⟨blank⟩

⟨proc stmt⟩ : ≡ ⟨function⟩ ⟨A⟩ ⟨B⟩ *or* ⟨id⟩ = : (⟨outlist⟩) ⟨B⟩ *or* ⟨id⟩ : (⟨succr list⟩)

⟨ppol⟩ : ≡ ⟨blank⟩ *or* ⟨ppol⟩ ⟨oe⟩,

⟨pol⟩ : ≡ ⟨ppol⟩ *or* ⟨pol⟩, *or* ⟨pol⟩, ⟨oe⟩

⟨A′⟩ : ≡ = : (⟨pol⟩)

⟨ppsl⟩ : ≡ ⟨blank⟩ *or* ⟨ppsl⟩ ⟨suc⟩,

⟨psl⟩ : ≡ ⟨ppsl⟩ *or* ⟨psl⟩, *or* ⟨psl⟩, ⟨suc⟩

⟨B′⟩ : ≡ : (⟨psl⟩)

⟨F*⟩ : ≡ ⟨function⟩ *or* ⟨pure function⟩ *or* ⟨id⟩

⟨A*⟩ : ≡ ⟨A⟩ *or* ⟨A′⟩

⟨B*⟩ : ≡ ⟨B⟩ *or* ⟨B′⟩

⟨pure procedure⟩ : ≡ ⟨pure function⟩ ⟨A*⟩ ⟨B*⟩ *or* ⟨F*⟩ ⟨A′⟩ ⟨B*⟩ *or* ⟨F*⟩ ⟨A*⟩ ⟨B′⟩

[a pure procedure may have any of the forms of a procedure statement but at least one position of one existing list must be empty: at least one input parameter position or one output position or one successor position].

3.49 **return** statements

⟨**return** stmt⟩ : ≡ **return**

3.51 **type** declarations

⟨id list⟩ : ≡ ⟨id⟩ *or* ⟨id list⟩, ⟨id⟩

⟨**bool** declar⟩ : ≡ **boolean** (⟨id list⟩)

⟨**integer** declar⟩ : ≡ **integer** (⟨id list⟩)

⟨**type** declar⟩ : ≡ ⟨**bool** declar⟩ *or* ⟨**integer** declar⟩

3.52 **array** declarations

⟨ulb⟩ : ≡ ⟨ar exp⟩
⟨ulb A⟩ : ≡ ⟨ar exp⟩
⟨ulbl⟩ : ≡ ⟨ulbA⟩ : ⟨ulb⟩
or ⟨ulbA⟩, ⟨ulbl⟩, ⟨ulb⟩
⟨**array** seg⟩ : ≡ ⟨id⟩ [⟨ulbl⟩] *or* ⟨id⟩, ⟨**array** seg⟩
⟨arl⟩ : ≡ ⟨**array** seg⟩ *or* ⟨arl⟩, ⟨**array** seg⟩
⟨**array** declar⟩ : ≡ **array** (⟨arl⟩)

3.53 **switch** declarations

⟨de⟩ : ≡ ⟨id⟩ *or* ⟨integer⟩ *or* ⟨id⟩ [⟨exp⟩]
⟨del⟩ : ≡ ⟨de⟩ *or* ⟨del⟩, ⟨de⟩
⟨**switch** declar⟩ : ≡ **switch** ⟨id⟩ := (⟨del⟩)

3.54 Function declarations

⟨function declar⟩ : ≡ ⟨id⟩ (⟨id list⟩) : = ⟨exp⟩

3.55 **comment** declarations

⟨not sc⟩ : ≡ ⟨any character other than ";"⟩
⟨str⟩ : ≡ ⟨not sc⟩ *or* ⟨str⟩ ⟨not sc⟩
⟨**comment** declar⟩ : ≡ **comment** ⟨str⟩

3.56 **common** declarations

⟨com entry⟩ : ≡ ⟨id⟩ (⟨id list⟩)
⟨com list⟩ : ≡ ⟨com entry⟩ *or* ⟨com list⟩,
⟨com entry⟩
⟨**common** declar⟩ : ≡ **common** (⟨com list⟩)

3.57 Statements and declarations

⟨b stmt⟩ : ≡ ⟨assnmt stmt⟩
or ⟨**go to** stmt⟩
or ⟨**do** stmt⟩
or ⟨**stop** stmt⟩
or ⟨proc stmt⟩ *or* ⟨**return** stmt⟩
⟨basic stmt⟩ : ≡ ⟨b stmt⟩ *or*
⟨label⟩ : ⟨b stmt⟩
⟨sl⟩ : ≡ ⟨stmt⟩ *or* ⟨sl⟩ ; ⟨stmt⟩
or ⟨sl⟩ ; ⟨declar⟩
or ⟨declar⟩ ; ⟨sl⟩
⟨u stmt⟩ : ≡ ⟨b stmt⟩ *or* ⟨**if** stmt⟩
or ⟨**for** stmt⟩ *or* ⟨**converge** stmt⟩
or **begin** ⟨sl⟩ **end**
⟨stmt⟩ : ≡ ⟨u stmt⟩ *or* ⟨label⟩ : ⟨u stmt⟩
⟨declar⟩ : ≡ ⟨**type** declar⟩ *or* ⟨**array** declar⟩
or ⟨**switch** declar⟩ *or* ⟨function declar⟩
or⟨**comment** declar⟩*or*⟨**common** declar⟩

3.6 *Syntactic properties of IAL statement lists*

a) L is the entry label of a statement S if S has the form:

L : ⟨u stmt⟩

b) If d_2 is a designational expression in statement list S and there is a **switch** declaration:

switch x_1 : = (···, d_2, ···)

then d_2 is said to *succeed* any designational expression of the expression of the form x_1 [⟨exp⟩]. Further, if d_3 succeeds d_2 and d_2 succeeds d_1, then d_3 succeeds d_1.

3.7 *Definition of an IAL program*

An IAL program is a statement list (⟨sl⟩) with the following properties:

a) No two statements or substatements have the same prefixed label.
b) No designational expression succeeds itself.
c) Every designational expression has some label as successor, or is itself a label.
d) To every n-place array name or subscripted variable name there corresponds one and only one entry of that name in an n-place array segment (n = 1/2 number of entries in the upper-lower-bound list, ⟨ulbl⟩).

4. Acknowledgment

The proposed International Algebraic Language is the direct and indirect product of many people's work. Most directly it is the result of the Zurich Conference at which the participants were: F. L. Bauer, H. Bottenbruch, H. Rutishauser, and K. Samelson (representing the GAMM), and J. W. Backus, C. Katz, A. J. Perlis, and J. H. Wegstein (representing the ACM). Extensive preparatory work was done by larger ACM and GAMM committees [1]. All of this, in turn, was based on prior work done by many individuals and groups in the field of automatic programming. The earliest work (to the author's knowledge) in this area was that of Rutishauser, in Switzerland, and of Laning and Zierler, in the United States.

5. Reference

[1] PERLIS, A. J., and K. Samelson: *Preliminary Report—International Algebraic Language.* Communications of the ACM, Vol. 1, No. 12, Dec. 1958.

6. Discussion

Hans Riesel (Sweden): According to Mr. Bauer the symbols and notations of ALGOL were chosen to be a compromise between the characters now available on coding and punching devices and those which are desirable. It would have been better to choose a few characters, so that almost everybody has punching devices which can use the language, and that translation from the Algol-reference language to a special language for the computer may be avoided. If it is necessary to make a translation, the symbols should be much closer to ordinary mathematical representation, supplemented if necessary for the special needs of numerical work.
Mr. Backus said that a good common language should be a good means of communication both amongst machines and also between people. There is a danger that the present compromise might prove to be neither.

Jan V. Garwick (Norway): In at least 90% of all cases, **for** statements are used in the following way:

1) There is an array declaration e.g. of the form

array (a [0 : 19])

2) There is a **for** statement, followed by another statement, of the form

for i : = 0 (1) 19
··· a [i] ···

Most of the information in the **for** statement is here redundant because the array declaration and the a [i] in the statement following the **for** together show the range of values of i.
It seems therefore better to use a statement of the form,

2a) **for all** i
··· a [i] ···

Generalisations of this **for all** statement could be:

for all $i < k$
for all $i \geq s$
for all $i \neq k$
for all $s \leq i < k$

(if here s = k, the following statement is ignored).

If one wants to count backwards, a dash is added to i, e.g.

for all $i' \geq s$

(meaning $i = 19, 18, \cdots s$).

ALGOL appears to lack any expressions for packing and unpacking and for masking words. If one wished to describe an Algol-computer in Algol, expressions of this type would seem to be necessary.

L. Nolin (France): Le vocabulaire minimum du langage ALGOL étant maintenant à peu près fixé, il semble opportun de songer sérieusement à le compléter en introduisant des abréviations couramment utilisées dans la rédaction des problèmes numériques.
Par exemple, soient ξ, α, β, γ, ν des expressions arithmétiques et Γ une expression booléenne: $\mu\ (\xi, \alpha, \beta, \gamma, \Gamma, \nu)$ est une expression arithmétique[1]) désignant:
— le plus petit nombre ξ défini à 10^{α} près, compris entre β et γ, qui satisfait à la condition Γ, s'il existe un tel nombre
— le nombre ν dans les autres cas; $\exists\ (\xi, \alpha, \beta, \gamma, \Gamma)$ est une expression booléenne signifiant:
— il existe au moins un nombre ξ, défini à 10^{a} près, compris entre β et γ, qui satisfait à la condition Γ; $\forall\ (\xi, \alpha, \beta, \gamma, \Gamma)$ est une expression booléenne signifiant:
— tout nombre ξ, défini à 10^{α} près, compris entre β et γ, satisfait à la condition Γ.
Il est clair que ces expressions sont dans chaque cas particulier, des abréviations d'expressions — fort longues — du langage ALGOL primitif.

[1]) Cette expression pourrait être utilisée, par exemple, dans un énoncé de consultation de table.