An Introduction to ALGOL 60

By M. Woodger

This is an account of the more important features of the algorithmic language ALGOL 60. The aim is to explain rather than to define the language, and for full details reference should be made to the Report published in *Numerische Mathematik*, Vol. 2, pp. 106–136 (1960).

1. Introduction

1.1 *Purpose*

ALGOL 60 is a formal symbolic language for expressing processes of computation (*algorithms*).

The order code for any individual computer is, in fact, such a language—it has a "syntax" or set of rules specifying what are meaningful combinations of symbols, and a "semantics" or set of rules specifying the meanings of these combinations, that is to say the action taken by the computer when the orders are executed.

Such "machine languages" are burdened with details of the particular structure, arithmetic facilities, input–output and storage arrangements of the computer concerned. The purpose of ALGOL 60 is to provide a language for the expression of computation processes to the extent that they are independent of these details, and in a form which can be translated automatically, into any particular machine code, by a suitable translator program written for the computer concerned. The "machine oriented" details of the computation are to be embodied in the translator rather than in the ALGOL 60 program.

A second use of ALGOL 60 is for expressing processes of computation in a form suitable for human consumption. Whereas an unbroken sequence of symbols of a few different sorts is satisfactory for input to a computer, legibility requires a display on the printed page, and a variety of forms of expression which will reflect the natural subdivision of the process. To accommodate these varying requirements there are three forms of ALGOL 60: it is defined in a form known as the *reference language* (employing a total of 116 basic symbols but in a linear fashion); it is used as input to translators in an appropriate *hardware representation* (employing, e.g., 5-bit characters on punched paper tape); and it is used in publications in the form of *publication language*. The latter admits the use of suffixes ($a_{i,j}$ corresponding to $a[i, j]$ in the reference language), exponents (a^3 corresponding to $a\uparrow 3$), Greek letters, spaces, and line groupings of characters. Apart from such relatively trivial differences the three forms of ALGOL 60 are identical in content.

Words printed in bold type such as **begin** represent single basic symbols. In the typescript of an ALGOL 60 program these would be underlined.

1.2 *Subject-Matter*

The computing processes described deal principally with ordinary real numbers, integers, and arrays of these. The real numbers are of necessity approximated digitally, but how this is done is not expressed.

In order to avoid reference to particular storage locations a notation is introduced for variable quantities in store, and it is important to be clear as to its precise significance. A letter, or a string of letters or decimal digits of which the first is a letter, is called an *identifier*. An identifier may be used in an ALGOL 60 program as a *simple variable*. This means that the program, when ultimately translated and run on a computer, will associate a particular storage location with that identifier. The number held in this store at any stage in the calculation is called the *current value* of the variable. An instruction or *statement* in the ALGOL 60 program which contains this identifier calls either for the use of this number in calculation or for its replacement by the result of calculation—the latter is referred to as "assignment of a value to the variable."

1.3 *Calculation Rules*

The course of the computation process described by the ALGOL 60 program may be visualized as a succession of assignments of values to variables. The program expresses rules for the calculation of these assigned values, and these rules take a variety of forms. The simplest rules are provided by ordinary algebraic expressions compounded from simple variables and numerical (decimal) constants by the usual symbols for addition, subtraction, multiplication and division. If E denotes such an expression, the instruction in ALGOL 60 which calls for the assignment of the current value of E to a variable V is written

$$V := E.$$

E may well contain V itself; for example, the assignment statement $V := V + 1$ increases the current value of V by 1.

To indicate that the variable V is to be treated as a real variable (say), rather than as an integer or other "type" of quantity, a *type declaration* **real** V is used. A succession of type declarations for the variables employed, followed by a succession of assignment statements, each statement and declaration being separated from its neighbours by a semicolon (;), and the whole being enclosed between the "statement brackets" **begin** and **end,** would constitute a simple ALGOL 60 program.

1.4 *Arrays*

Large blocks of numerical data are commonly treated as *arrays*, the elements of which are specified by sets of integers. The number of integers in such a set is the

"dimensionality" of the array. Thus a matrix A is a two-dimensional array for which a set of two integers, the row number i and column number j, is used to single out a particular element. This element is denoted by the suffixed symbol A_{ij}. If the elements of a matrix A of n columns are stored in consecutive locations row by row, the position number of element A_{ij} relative to A_{11} as number 0 is $(i-1)n+(j-1)$, and this rule for finding the (i, j) element must be available to the computer whenever the value of this element is called for.

An array is in effect a function of a number of integer variables, the values of the function being explicitly listed (stored) and the values of the variables being used to locate the function value in the list (store). This may be contrasted with the situation with an algebraic expression, which is effectively a function of the variables it contains. In that case no function values are stored, but the values of the variables are used to calculate the function value by a formula. A formula can generally be used for arbitrary values of the arguments. An array, on the other hand, is limited, for reasons of storage, in each of its dimensions. ALGOL 60 deals only with arrays whose dimensions are independent, i.e. generalized rectangular arrays, and in particular only those employing consecutive integer suffix values. The extent of an array is specified by the lower and upper limiting values of each of its suffixes, and this information for each array, together with the type of its elements, is provided by an *array declaration*.

Each element of an array may be assigned a value independently of the others. The general form of such an assignment statement is

$$V[E, E, \ldots, E] := F$$

or in "publication language" ALGOL 60 (which permits the use of suffixes instead of square brackets)

$$V_{E, E, \ldots, E} := F.$$

Here V is a *subscripted variable* (the identifier naming the array); each E (a *subscript*) is an expression whose current value is used to locate the appropriate element of the array, and F is the expression whose current value is to be assigned to that element.

1.5 *Constituents of a Program*

The basic constituents of an ALGOL 60 program are thus *statements* which are executed as instructions in the order in which they are written and have the effect of assigning values to certain variables, and *declarations* which are not themselves executed as instructions but provide information necessary to the execution of the statements following them.

There is a third (unnamed) category of constituents which corresponds to the control or "red tape" instructions in machine language. These make it possible

(i) to break off a sequence of calculations and start again somewhere else in the program (the "jump" or *go to* statements);

(ii) to skip certain statements in a sequence if certain conditions are not satisfied (*conditional* statements); and

(iii) to repeat the next statement for a succession of values of a variable (the *for* statements).

Since no reference is made to storage locations of the ALGOL 60 statements themselves, they have to be *labelled* to be identified as destinations for *go to* statements. Any identifier or unsigned integer may be used as a label, written in front of the statement labelled and separated from it by a colon (:).

Sequences of statements may be combined within the statement brackets **begin** and **end** to form *compound statements*, which again may be labelled.

1.6 *Blocks and Declarations*

Each identifier used in an ALGOL 60 program, other than as a label, is introduced by a declaration which gives information concerning it, and which is referred to by the translator when executing the statements in which the identifier appears. This declaration is written (possibly with others) following the **begin** symbol of some compound statement (possibly the whole program). A compound statement containing declarations in this way is called a *block*, and each declaration is valid only for the block to which it is attached. This means that each identifier I is "local" to the block B for which it is declared, in the sense that on exit from B either via the **end** (on completion of the last statement in B) or on execution of a *go to* (jump) statement leading outside B, I has no longer the declared significance and may be used afresh in a new declaration to denote some entirely different thing. Alternatively, if the block B is itself a component statement of a larger block A for which I was already declared, then, while B is being executed, I has the local significance declared for B but on exit to A it reverts to the significance it had when B was entered, its value (if a variable) remaining unaffected by passage through B. Identifiers used in A and not declared for B retain their significance within B. Thus every block automatically introduces a new level of nomenclature.

Labels are automatically "local" to the blocks in which they are used, so that a jump into a block from outside is not possible—that would bypass the governing declarations.

A declaration D, attached to a block B and governing an identifier I representing a variable or array, may be prefixed with the symbol **own.** This has the effect that on re-entering B the value of the variable (or values of the array elements) is as it was left at the previous exit from B.

1.7 *Functions and Procedures*

We have noted that an algebraic expression can be considered as a rule for evaluating a function, the constituent variables in the expression being the arguments of the function. More generally, the same is true of any ALGOL 60 program if we single out one of the variables

to which the program assigns a value as being the value of the function. ALGOL 60 provides for the definition of a function in this way, giving it a name and indicating the variables concerned, and for using its name (followed by a parenthesized list of expressions to be used as its arguments) as a constituent of algebraic expressions elsewhere, i.e. as a "function designator." A notation is also provided for defining as a "procedure" any ALGOL 60 program even when its effect is not simply the assignment of a value to a single variable. In this case the procedure name (followed by a parenthesized argument list as before) appears not as part of an algebraic expression but as a statement—a "procedure statement"—in the program. Both functions and procedures are introduced and defined by *procedure declarations* which comprise the defining block of program or statement (the "procedure body") prefixed by the symbol **procedure,** the name of the procedure (any identifier), and details regarding which identifiers in it are to be treated as the arguments or "parameters" and how they are to be used when the procedure is called. A type declaration in front of the symbol **procedure** indicates that a function is being defined and gives the type of its value.

For example the procedure declaration

real procedure *sumsq* (P, Q, R, S);
$sumsq := P^2 + Q^2 + R^2 + S^2$

might be used to define the function designator appearing in a statement such as

$y := T^2 + sumsq\,(a - m, b - m, c - m, d - m)/3.$

When this statement is executed the effect is as if *sumsq* represented a simple variable to which a value was first assigned by *substituting* the "actual parameters" $a - m$, $b - m$, $c - m$, $d - m$ respectively for the "formal parameters" P, Q, R, S in the procedure body and then executing the resulting assignment statement. $T^2 + sumsq/3$ is then calculated and its value assigned to y.

It should be noted that the letters P, Q, R, S (in general any identifiers) bear no relation to other identifiers outside the procedure body and do not represent "variables"—they are used simply to mark positions for substitution in the procedure body, and for this reason are called "formal parameters."

A simple example of a procedure statement is

$TEST\,(b^2 - 4 \times a \times c, L1, L2, L3)$

which might be used as a three-way discrimination using current values of variables a, b, c to continue the program at statements labelled $L1$, $L2$ or $L3$ according as $b^2 - 4 \times a \times c$ is positive zero or negative. The procedure declaration defining *TEST* (which must appear in the head of a block in which the above statement appears) could be as follows:

procedure *TEST* (a, P, Z, N); **if** $a > 0$ **then go to** P **else if**

$a = 0$ **then go to** Z **else go to** N.

This illustrates also a simple form of conditional statement. If B represents some condition which may or may not be currently satisfied, and S, T represent statements, the effect of

if B **then** S **else** T

is the same as that of S if B holds, and is the same as that of T if B does not hold. The statement

if B **then** S

has no effect (is skipped) if B does not hold.

1.8 *Standard Functions*

It is recommended in the Report that the following identifiers should be reserved for standard functions which may be available with a particular translator without explicit declaration, through the use of a library of subroutines. x denotes the current value of the expression E.

abs (E)	the modulus (absolute value) of x
sign (E)	$+1$ if $x > 0$; 0 if $x = 0$; -1 if $x < 0$ } of type **integer**
entier (E)	the integral part of x (largest integer not greater than x), of type **integer**
sqrt (E)	square root of x
sin (E)	sine of x
cos (E)	cosine of x
arctan (E)	principal value of arctangent of x
ln (E)	natural logarithm of x
exp (E)	exponential function of x.

1.9 *Input and Output*

The input of data and the output of results have been excluded from detailed consideration in the reference language, but it is intended that suitable procedures be available with individual translators which will perform these functions. These procedures would be referred to by name in the ALGOL program and might require as parameters information concerning the layout of data on the external medium, i.e. information not expressible in the language. For this purpose arbitrary strings of ALGOL 60 symbols may be entered as actual parameters of procedures through the use of two special "string quotes" symbols, and the body of a procedure declaration may be expressed in machine code while the heading remains expressed in ALGOL 60 for reference.

2. Examples

Before indicating the full generality of the various categories of symbolism provided in ALGOL 60 we give first some elementary examples.

2.1 The following procedure declaration defines *max* (a, n) as the function whose arguments are a vector (one-dimensional array) a of n elements, and the integer n, and whose value is the modulus of the largest component of the vector.

```
    real procedure max (a, n);
L1: begin real m; m := 0; L2: for i := 1 step 1 until
      n do
L3: begin real x; x := a_i; if x < 0 then x := -x;
      if x > m then m := x end;
L4: max := m end.
```

This illustrates a simple kind of "for" statement—the statement labelled *L2* causes the statement *L3* to be executed *n* times with *i* taking the values 1, 2, . . . *n* in turn, before proceeding to statement *L4*. The labels are superfluous in this case since no *go to* statement is used.

It is worth noting that one cannot eliminate *m* and write the procedure body simply as

```
L1: begin max := 0; L2: for i := 1 step 1 until n do
L3: begin real x; x := a_i; if x < 0 then x := -x;
      if x > max then max := x end end
```

because wherever the identifier *max* appears other than on the left of an assignment statement—in this case in the condition ($x >$ *max*)—it calls into use the procedure *max* itself, which is not the intention here.

2.2 To illustrate the use of the above procedure declaration by a function designator in an algebraic expression, suppose we have a matrix *B* of *r* rows and *c* columns and we wish to normalize its rows to have largest elements unity (where the rows have at least one non-zero element). We could write:

```
    for i := 1 step 1 until r do
1:  begin  array a [1 : c]; for j := 1 step 1 until c do
       a_j := B_{i,j};
    for j := 1 step 1 until c do
    if max (a, c) ≠ 0 then B_{i,j} := B_{i,j}/max (a, c) end.
```

Here an array *a* local to the block labelled 1 has been introduced and a copy of the current *i*th row of *B* assigned to it simply in order to have a one-dimensional array to use as actual parameter of *max*. *a* is defined by the array declaration **array** *a* [1 : *c*] as being one-dimensional with suffix values ranging from 1 to *c*, and because no "type" is indicated, the type **real** is understood (a special convention for array declarations). *B* could have been introduced by a declaration of the form

array *B* [1 : *r*, 1 : *c*].

2.3 The following example from the ALGOL 60 report illustrates a more general form of the heading of a procedure declaration.

```
procedure Innerproduct (a, b) order: (k, p) Result: (y);
value k;
integer k, p; real y, a, b;
begin  real s; s := 0;
for p := 1 step 1 until k do s := s + a × b;
y := s end Innerproduct
```

The formal parameters of this procedure are *a*, *b*, *k*, *p*, *y*, and could have simply been listed between a single pair of parentheses in the first line. The convention used here is that any separating comma ("parameter delimiter") may be replaced without effect on the program by a string of letters followed by a colon and enclosed between reversed parentheses thus

) *order:* (

By this means the various parameters may more easily be recognized by the reader.

The last line illustrates a similar convention that the symbol **end** may be followed, without effect, by any sequence of symbols not containing **end** or **else** or a semicolon.

The second line of the declaration is known as the "value part," and has been absent from the examples given earlier. It indicates those parameters which are to be "called by value" rather than "called by name" as hitherto. This means that at a procedure call these formal parameters in the procedure body are *not* to be replaced by the actual parameters but are to be treated in the execution of the procedure as if they were identifiers local to the procedure body, representing variables or arrays to which were initially assigned the current values of the actual parameter expressions.

The third line of the declaration is called the "specification part" and provides in general information concerning the kinds and types of admissible actual parameters.

In this example the "running variable" *p* of the "for" statement has deliberately been made one of the parameters so that it may also appear as a suffix in the actual parameter expressions substituted for *a* and *b*. Thus a particular procedure statement employing this procedure to form the inner product of a vector *B* of order 10 and a vector defined by fixing the first and third suffix of a three-dimensional array *A* might be

Innerproduct (*A* [*t*, *P*, *u*], *B* [*P*], 10, *P*, *Y*)

Of course the procedure could also be abused (presumably successfully) by writing *Innerproduct* (*C*, *D*, *n*, *i*, *E*) to have the same effect as

$$E := n \times C \times D$$

in which *C*, *D* and *n* are simple variables, the last a positive integer.

By omitting the parameter *y*, replacing the last assignment statement by *Innerproduct* := *s*, and preceding the symbol **procedure** by **real** this procedure declaration would be changed into a definition of *Innerproduct* as a function designator.

3. Syntax

In order to describe precisely what combinations of symbols are meaningful in ALGOL 60, a special notation is used which is best explained by an example. Given that the symbol ⟨letter⟩ names any one of the fifty-two lower or upper case Latin letters, and that ⟨digit⟩ names

any one of 0, 1, . . . 9, we can define the name ⟨identifier⟩ by the formula

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩ | ⟨identifier⟩ ⟨digit⟩

in which the vertical strokes may be read "or." This is to be read as saying that an identifier is either a letter or is an identifier followed by a letter or is an identifier followed by a digit. It defines the notion of identifier recursively. Thus *V*23*a* is an identifier because *V*23 is one, *V*23 because *V*2 is, and *V*2 because the single letter *V* is. This corresponds with the verbal definition in Section 1.2 above as "a letter or a string of letters or decimal digits of which the first is a letter."

In general, the juxtaposition of names in these "metalinguistic formulae" is used to name the juxtaposition of the sequences of symbols named, and any ALGOL 60 symbol appearing is used to name itself. Thus examples of sequences named by +⟨digit · digit⟩ would be +3·7, +0·0, +9·9.

In the next section the full extent of the more important means of expression in ALGOL 60 will become apparent.

4. Syntax of Expressions

4.1 *Numbers*

Constant numbers are expressed in ALGOL 60 in the decimal notation with sign, integral part, decimal point, fractional part, and finally a signed power of ten to be used as a scale factor. The exponent of 10 is brought down to the level of the text by using a special suffix symbol $_{10}$ for the radix, and this and various parts of the general form may be omitted when superfluous, e.g. $-1\cdot083_{10}-02$, $\cdot7300$, $+_{10}7$.

The precise definition in several steps is as follows:

⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨unsigned integer⟩ ⟨digit⟩
⟨integer⟩ ::= ⟨unsigned integer⟩ | +⟨unsigned integer⟩ | −⟨unsigned integer⟩
⟨decimal fraction⟩ ::= ·⟨unsigned integer⟩
⟨exponent part⟩ ::= $_{10}$⟨integer⟩
⟨decimal number⟩ ::= ⟨unsigned integer⟩ | ⟨decimal fraction⟩ | ⟨unsigned integer⟩⟨decimal fraction⟩
⟨unsigned number⟩ ::= ⟨decimal number⟩ | ⟨exponent part⟩ | ⟨decimal number⟩⟨exponent part⟩
⟨number⟩ ::= ⟨unsigned number⟩ | +⟨unsigned number⟩ | −⟨unsigned number⟩

4.2 *Variables*

⟨simple variable⟩ ::= ⟨identifier⟩
⟨array identifier⟩ ::= ⟨identifier⟩

Although strictly speaking redundant, these definitions help to convey part of the meaning at the same time as the structure is defined.

⟨subscript list⟩ ::= ⟨arithmetic expression⟩ | ⟨subscript list⟩, ⟨arithmetic expression⟩
⟨subscripted variable⟩ ::= ⟨array identifier⟩ [⟨subscript list⟩]
⟨variable⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩

4.3 *Function Designators*

⟨actual parameter⟩ ::= ⟨string⟩ | ⟨expression⟩ | ⟨array identifier⟩ | ⟨switch identifier⟩ | ⟨procedure identifier⟩

Strings are not discussed in this article. Switches are defined under 4.7.

⟨letter string⟩ ::= ⟨letter⟩ | ⟨letter string⟩⟨letter⟩
⟨parameter delimiter⟩ ::= , |)⟨letter string⟩: (
⟨actual parameter list⟩ ::= ⟨actual parameter⟩ | ⟨actual parameter list⟩ ⟨parameter delimiter⟩ ⟨actual parameter⟩
⟨procedure identifier⟩ ::= ⟨identifier⟩
⟨function designator⟩ ::= ⟨procedure identifier⟩ | ⟨procedure identifier⟩ (⟨actual parameter list⟩)

4.4 *Simple Arithmetic Expressions*

⟨adding operator⟩ ::= + | −
⟨multiplying operator⟩ ::= × | / | ÷
⟨primary⟩ ::= ⟨unsigned number⟩ | ⟨variable⟩ | ⟨function designator⟩ | (⟨arithmetic expression⟩)
⟨factor⟩ ::= ⟨primary⟩ | ⟨factor⟩ ↑ ⟨primary⟩
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩⟨multiplying operator⟩⟨factor⟩
⟨simple arithmetic expression⟩ ::= ⟨term⟩ | ⟨adding operator⟩⟨term⟩ | ⟨simple arithmetic expression⟩⟨adding operator⟩⟨term⟩

The operators +, −, × have the conventional meaning and yield expressions of type **integer** when both operands have type **integer,** otherwise **real.**

The operator / denotes division, to be understood as a multiplication of the preceding term by the reciprocal of the following factor, the result having always type **real.**

The operator ÷ also denotes division, but is restricted to operands of type **integer** and yields the integral part of the quotient, defined by

$$m \div n = sign\,(m/n) \times entier\,(abs\,(m/n))$$

The operator ↑ denotes exponentiation, the preceding factor being the base and the following "primary" the exponent. If the exponent is a positive integer *i* the result is a product of *i* equal factors of the same type as the base, if a negative integer then the result is the reciprocal of such a product and of type **real,** or is undefined if the base is zero, while if the exponent is zero the result is unity of the same type as the base. If the exponent *r* is of type **real** and the base *a* is positive the result is $exp\,(r \times ln\,(a))$ of type **real**; if *a* is zero the result is zero of type **real** for positive *r*, and otherwise is undefined.

The sequence of operations in evaluating an expression is from left to right subject to the usual use of parentheses and to the following rules of precedence between the operators, which are reflected in the syntax above:

first: $\uparrow$
second: $\times$ / $\div$
third: $+$ $-$

4.5 *Boolean Expressions*

These are used chiefly to express conditions for use in conditional statements and, as introduced below, in conditional expressions. For these purposes a condition is essentially a two-valued thing, it either holds or does not hold, and one can manipulate conditions as two-valued variables by the algebra of logic, i.e. Boolean algebra, which is available as part of ALGOL 60. A variable may be declared to have type **Boolean,** and its value is then either **true** or **false** (these being basic symbols). Such a variable may be assigned the current value of a Boolean expression just as in ordinary arithmetic. A function designator may have type **Boolean,** and one can have a Boolean array.

The simplest conditions are relations of equality or inequality between arithmetic expressions. From any condition we can derive the reverse condition by preceding it by the logical negation sign $\neg$, and from any two conditions we can derive, by the logical operators $\wedge$ (and), $\vee$ (or), $\supset$ (implies) and $\equiv$ (equivalent), compound conditions expressing respectively that both hold, that at least one holds, that either the first does not hold or the second does or both, and that they either both hold or both do not hold.

For example:

if $(a > b + 1) \vee (a < b - 1)$ **then** **go to** L.

This is equivalent to

if $\neg$ *abs* $(a - b) \leqslant 1$ **then** **go to** L

and also to

$B :=$ *abs* $(a - b) \leqslant 1$; **if** $\neg B$ **then** **go to** L

which illustrates the "calculation" of a condition.

The sequence of operations in evaluating a Boolean expression follows the rules for arithmetic expressions, with the rules of precedence extended thus:

4th: $< \leqslant = \geqslant > \neq$
5th: $\neg$
6th: $\wedge$
7th: $\vee$
8th: $\supset$
9th: $\equiv$

so the condition in the first example above could be written without parentheses as $a > b + 1 \vee a < b - 1$.

The syntax built up following this order of precedence is as follows:

⟨relational operator⟩ ::= $< \mid \leqslant \mid = \mid \geqslant \mid > \mid \neq$

⟨relation⟩ ::= ⟨arithmetic expression⟩⟨relational operator⟩⟨arithmetic expression⟩
⟨logical value⟩ ::= **true** | **false**
⟨Boolean primary⟩ ::= ⟨logical value⟩ | ⟨variable⟩ | ⟨function designator⟩ | ⟨relation⟩ | (⟨Boolean expression⟩)
⟨Boolean secondary⟩ ::= ⟨Boolean primary⟩ | $\neg$ ⟨Boolean primary⟩
⟨Boolean factor⟩ ::= ⟨Boolean secondary⟩ | ⟨Boolean factor⟩ $\wedge$ ⟨Boolean secondary⟩
⟨Boolean term⟩ ::= ⟨Boolean factor⟩ | ⟨Boolean term⟩ $\vee$ ⟨Boolean factor⟩
⟨implication⟩ ::= ⟨Boolean term⟩ | ⟨implication⟩ $\supset$ ⟨Boolean term⟩
⟨simple Boolean⟩ ::= ⟨implication⟩ | ⟨simple Boolean⟩ $\equiv$ ⟨implication⟩

4.6 *Conditional Expressions*

These provide a method (due to J. McCarthy) for selecting one of a sequence of expressions $E_1, E_2, \ldots$ according to the current values of a corresponding sequence of conditions (Boolean expressions) $B_1, B_2, \ldots$ If B_k is the first condition to hold, i.e. to have the value **true,** then the value of the following conditional arithmetic expression is that of E_k, regardless of whether or not $B_{k-1}, B_{k+2}, \ldots$, or $E_{k-1}, E_{k+2}, \ldots$ are even defined.

if B_1 **then** E_1 **else if** B_2 **then** E_2 **else** $\ldots$ **else** E_n

In particular $E_1, E_2, \ldots$ may themselves be Boolean expressions.

This enables us to complete the syntax of arithmetic and Boolean expressions as follows:

⟨if clause⟩ ::= **if** ⟨Boolean expression⟩ **then**
⟨Boolean expression⟩ ::= ⟨simple Boolean⟩ | ⟨if clause⟩⟨simple Boolean⟩ **else** ⟨Boolean expression⟩
⟨arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩ | ⟨if clause⟩⟨simple arithmetic expression⟩ **else** ⟨arithmetic expression⟩

Discontinuous functions are naturally expressed as conditional expressions. For example

sign (E) is equivalent to

if $E > 0$ **then** 1 **else if** $E = 0$ **then** 0 **else** -1

and *abs* (E) is equivalent to

if $E < 0$ **then** $-E$ **else** E

4.7 *Switches, Designational Expressions*

Just as an arithmetic expression is a rule for obtaining a numerical value, so a designational expression is a rule for obtaining a designation of a statement, i.e. a label. The only such expression so far introduced is simply a label itself, but we now provide a notation—the "switch"—for using any arithmetic expression whose value is a positive integer to select one from a list of labels (or again in general designational expressions). This is in some ways analogous to a one-dimensional array. Whereas an array declaration merely specifies size, shape and type of values, a *switch declaration*

actually exhibits the list of values. The designational expression whose value is the kth label in this list is written like a suffixed variable with k as suffix and the switch identifier as identifier.

For example, if the switch S were defined by the declaration

switch $S := L, P, 4, L$

then the "switch designators" $S[1]$ and $S[4]$ and $S[3 + 1 \uparrow 1]$ have each as value the label L. These may themselves appear in *go to* statements within the block to which the switch declaration is attached, or as entries in other switch declarations.

Finally, one can have conditional designational expressions just as in the case of arithmetic and Boolean expressions, and the full syntax is as follows:

⟨label⟩ ::= ⟨identifier⟩ | ⟨unsigned integer⟩
⟨switch identifier⟩ ::= ⟨identifier⟩
⟨switch designator⟩ ::= ⟨switch identifier⟩[⟨arithmetic expression⟩]
⟨simple designational expression⟩ ::= ⟨label⟩ | ⟨switch designator⟩ | (⟨designational expression⟩)
⟨designational expression⟩ ::= ⟨simple designational expression⟩ | ⟨if clause⟩⟨simple designational expression⟩ **else** ⟨designational expression⟩
⟨expression⟩ ::= ⟨arithmetic expression⟩ | ⟨Boolean expression⟩ | ⟨designational expression⟩

5. Syntax of Statements

5.1 *Assignment Statements*

The only new point here is the provision for simultaneous assignments such as

$$a[b[1, 1]] := x := b[1, 1] := 3 \times x$$

which assigns the current value of $3 \times x$ to a certain element of the one-dimensional array a (depending on the current value of $b_{1,1}$), to x itself, and then to $b_{1,1}$. (All variables here must be integers.)

If the expression assigned is E of type **real,** the variables may be of type **integer** and then the value assigned to them is that of *entier* $(E + 0{\cdot}5)$.

⟨left part⟩ ::= ⟨variable⟩ :=
⟨left part list⟩ ::= ⟨left part⟩ | ⟨left part list⟩⟨left part⟩
⟨assignment statement⟩ ::= ⟨left part list⟩⟨arithmetic expression⟩ | ⟨left part list⟩⟨Boolean expression⟩

5.2 *Go to Statements*

⟨go to statement⟩ ::= **go to** ⟨designational expression⟩

5.3 *Dummy Statements*

⟨dummy statement⟩ ::= ⟨empty⟩
⟨empty⟩ names the "empty" sequence of symbols.

A dummy statement is simply a blank—nothing is written. Its only purpose is to place a label. A *go to* statement which is simply to lead out of a compound statement to the following one can refer to a labelled dummy statement before the **end,** thus:

begin . . .; **if** $x < 0$ **then go to** END; . . .; END: **end.**

5.4 *Procedure Statements*

These are syntactically identical with function designators (4.3 above) but are used as statements instead of as expressions.

⟨procedure statement⟩ ::= ⟨procedure identifier⟩ | ⟨procedure identifier⟩(⟨actual parameter list⟩)

The actual parameter list might be absent, e.g. in the case of a procedure which simply sounded an alarm; the procedure itself would then be in machine code.

5.5 *For Statements*

A "for clause" causes the statement S which it precedes to be executed zero or more times, and performs an assignment of a value to a "controlled variable" V immediately before each execution. The "for list" defines these consecutively assigned values. An element of this list which is an arithmetic expression causes the assignment of its current value to V and a single execution of S. An element of the form A**step**B**until**C where A, B, C are arithmetic expressions causes the assignment of the current values of A, $A + B$, $A + 2B$, . . . to V and corresponding execution of S, the operation terminating as soon as $V - C$ has the same sign as B (this test being made just prior to each execution, so that S is never executed if initially $A - C$ and B have equal signs). An element of the form E**while**F where E is an arithmetic expression and F a Boolean expression causes the assignment of the current value of E prior to each execution, this being repeated until the current value of F is **false,** the test being made prior to each execution as above. This additional facility enables the number of times a loop is executed to be made to depend on the results of calculation without the necessity of writing a special test instruction.

If S is left by a *go to* statement, interrupting the execution of the *for* statement, the then current value of V continues to be available outside; otherwise V is treated as local to the *for* statement.

⟨for list element⟩ ::= ⟨arithmetic expression⟩ | ⟨arithmetic expression⟩ **step** ⟨arithmetic expression⟩ **until** ⟨arithmetic expression⟩ | ⟨arithmetic expression⟩ **while** ⟨Boolean expression⟩
⟨for list⟩ ::= ⟨for list element⟩ | ⟨for list⟩, ⟨for list element⟩
⟨for clause⟩ ::= **for** ⟨variable⟩ := ⟨for list⟩ **do**
⟨for statement⟩ ::= ⟨for clause⟩⟨statement⟩ | ⟨label⟩ : ⟨for statement⟩

The last definition shows that *for* statements may be repeatedly labelled. This is true of all kinds of statement, as will be seen.

5.6 *Conditional Statements, Compound Statements, Blocks*

The following completes the syntax of statements:

⟨basic statement⟩ ::= ⟨assignment statement⟩ | ⟨go to statement⟩ | ⟨dummy statement⟩ | ⟨procedure statement⟩ | ⟨label⟩ : ⟨basic statement⟩
⟨unconditional statement⟩ ::= ⟨basic statement⟩ | ⟨for statement⟩ | ⟨compound statement⟩ | ⟨block⟩
⟨if statement⟩ ::= ⟨if clause⟩⟨unconditional statement⟩ | ⟨label⟩ : ⟨if statement⟩
⟨conditional statement⟩ ::= ⟨if statement⟩ | ⟨if statement⟩ **else** ⟨statement⟩
⟨statement⟩ ::= ⟨unconditional statement⟩ | ⟨conditional statement⟩
⟨compound tail⟩ ::= ⟨statement⟩ **end** | ⟨statement⟩; ⟨compound tail⟩
⟨compound statement⟩ ::= **begin** ⟨compound tail⟩ | ⟨label⟩ : ⟨compound statement⟩
⟨block head⟩ ::= **begin** ⟨declaration⟩ | ⟨block head⟩; ⟨declaration⟩
⟨block⟩ ::= ⟨block head⟩; ⟨compound tail⟩ | ⟨label⟩ : ⟨block⟩

6. Syntax of Declarations

⟨declaration⟩ ::= ⟨type declaration⟩ | ⟨array declaration⟩ | ⟨switch declaration⟩ | ⟨procedure declaration⟩

6.1 *Type Declarations*

⟨type list⟩ ::= ⟨simple variable⟩ | ⟨simple variable⟩, ⟨type list⟩
⟨type⟩ ::= **real** | **integer** | **Boolean**
⟨local or own type⟩ ::= ⟨type⟩ | **own** ⟨type⟩
⟨type declaration⟩ ::= ⟨local or own type⟩⟨type list⟩

6.2 *Array Declarations*

⟨lower bound⟩ ::= ⟨arithmetic expression⟩
⟨upper bound⟩ ::= ⟨arithmetic expression⟩
⟨bound pair⟩ ::= ⟨lower bound⟩ : ⟨upper bound⟩
⟨bound pair list⟩ ::= ⟨bound pair⟩ | ⟨bound pair list⟩, ⟨bound pair⟩
⟨array segment⟩ ::= ⟨array identifier⟩[⟨bound pair list⟩] | ⟨array identifier⟩, ⟨array segment⟩
⟨array list⟩ ::= ⟨array segment⟩ | ⟨array list⟩, ⟨array segment⟩
⟨array declaration⟩ ::= **array** ⟨array list⟩ | ⟨local or own type⟩ **array** ⟨array list⟩

Examples

array *a,b,c*[7:*n*,2:*m*],*s,u*[−2 × *r*:10]

declares three matrices *a*, *b*, *c* and two vectors *s* and *u*.

own Boolean array *peter*3[1 + **if** *n* ⩾ 0 **then** *n* **else** 0: 20]

declares a vector called *peter*3 with elements each either **true** or **false,** the definition of the lower bound of its suffix involving a conditional arithmetic expression.

6.3 *Switch Declarations*

⟨switch list⟩ ::= ⟨designational expression⟩ | ⟨switch list⟩, ⟨designational expression⟩
⟨switch declaration⟩ ::= **switch** ⟨switch identifier⟩ := ⟨switch list⟩

6.4 *Procedure Declarations*

⟨formal parameter⟩ ::= ⟨identifier⟩
⟨formal parameter list⟩ ::= ⟨formal parameter⟩ | ⟨formal parameter list⟩⟨parameter delimiter⟩⟨formal parameter⟩
⟨formal parameter part⟩ ::= ⟨empty⟩ | (⟨formal parameter list⟩)
⟨identifier list⟩ ::= ⟨identifier⟩ | ⟨identifier list⟩, ⟨identifier⟩
⟨value part⟩ ::= ⟨empty⟩ | **value** ⟨identifier list⟩;
⟨specifier⟩ ::= **string** | ⟨type⟩ | **array** | ⟨type⟩ **array** | **label** | **switch** | **procedure** | ⟨type⟩ **procedure**
⟨specification part⟩ ::= ⟨empty⟩ | ⟨specifier⟩⟨identifier list⟩; | ⟨specification part⟩⟨specifier⟩⟨identifier list⟩;
⟨procedure heading⟩ ::= ⟨procedure identifier⟩⟨formal parameter part⟩; ⟨value part⟩⟨specification part⟩
⟨procedure body⟩ ::= ⟨statement⟩ | ⟨code⟩
⟨procedure declaration⟩ ::= **procedure** ⟨procedure heading⟩⟨procedure body⟩ | ⟨type⟩ **procedure** ⟨procedure heading⟩⟨procedure body⟩

7. Classification of Basic Symbols

As stated earlier the Reference language embraces 116 basic symbols.

⟨basic symbol⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨logical value⟩ | ⟨delimiter⟩
⟨delimiter⟩ ::= ⟨operator⟩ | ⟨separator⟩ | ⟨bracket⟩ | ⟨declarator⟩ | ⟨specificator⟩
⟨operator⟩ ::= ⟨arithmetic operator⟩ | ⟨relational operator⟩ | ⟨logical operator⟩ | ⟨sequential operator⟩
⟨arithmetic operator⟩ ::= + | − | × | / | ÷ | ↑
⟨logical operator⟩ ::= ≡ | ⊃ | ∨ | ∧ | ¬
⟨sequential operator⟩ ::= **go to** | **if** | **then** | **else** | **for** | **do**
⟨separator⟩ ::= , | . | $_{10}$ | : | ; | := | ␣ | **step** | **until** | **while** | **comment**
⟨bracket⟩ ::= (|) | [|] | ‘ | ’ | **begin** | **end**
⟨declarator⟩ ::= **own** | **Boolean** | **integer** | **real** | **array** | **switch** | **procedure**
⟨specificator⟩ ::= **string** | **label** | **value**

The separator ␣ denoting a space and the brackets ‘ and ’ are used in forming strings used as parameters of procedures. The separator **comment** is used to introduce explanatory text in an ALGOL 60 program without affecting the meaning of the program; the convention is used that

; **comment** ⟨any sequence of basic symbols not containing a semicolon⟩;

is equivalent to a single semicolon.

8. Example

procedure *Bisection*(*F*)*initial*: (*x*1,*d*1)*precision*: (*d*0) *result bounds*: (*xP*,*xL*);
comment *Finds bounds xP and xL (with difference less than d*0*) for a zero of the function F*(*x*) *at which its derivative is positive. Evaluates F*(*x*) *at x*1, *x*1 + *d*1 *and then at equispaced values of x until the zero is passed (indicated by b* = **true**) *when it is located by the method of repeated bisection, xP being the final value and xL the previous value the other side of the zero;* **value** *d*0; **real** *xL*, *xP*; **real procedure** *F*; **begin Boolean** *a*, *b*, *c*; **real** *x*, *d*;

a := *b* := **false**; *x* := *x*1; *d* := *d*1;
A: *c* := *sign*(*F*(*x*)) = *sign*(*d*);
b := *c*∧*a*∨*b*;
if *c*∧*a* **then** *xL* := *x*−*d*;
a := **true**;
if *b* **then** *d* := *d*/2;
if *c* **then** *d* := − *d*;
if *abs*(*d*) ⩾ *d*0 **then begin** *x* := *x*+*d*; **go to** *A* **end**;
xP := *x* **end** *Bisection*

This process has been used for non-linear eigenvalue problems in which the largest part of the calculation is the evaluation of the function $F(x)$, and this function involves empirical data so that its derivative is not available. The Boolean variable a is used to distinguish the first passage through the sequence of instructions (when it has the value **false**). The formula for the Boolean variable b is equivalent to $b := (c \wedge a) \vee b$ by the precedence rules for the logical operators. This shows that once b has been assigned the value **true** it will "stick" at that value—indicating that the required zero of $F(x)$ has been "bracketed." The part $c \wedge a$ ensures that this will not happen on the first step (when insufficient information is available for this decision). c = **true** indicates (when a is **true**) that addition of d to x will move it farther from the required zero, i.e. that the sign of d is to be changed.

Acknowledgement

This article is published with the permission of the Director of the National Physical Laboratory.

Information Processing

As we go to press with this edition of THE COMPUTER JOURNAL, the prepaid subscribers, who attended the International Conference on Information Processing at UNESCO in Paris, 15–20 June 1959, have received from the publishers their copies of the *Proceedings*.

This volume, which we, or *The Computer Bulletin*, may review in more detail later, has been published jointly by UNESCO, Paris; R. Oldenbourg Verlag, Munich; and Butterworths Scientific Publications, London. It includes a full report of the inaugural addresses at the Sorbonne and the closing speech by Professor Howard H. Aiken. These are rendered in French or English, as given at the time. There is an Introduction by UNESCO's chief editor explaining that despite the early publication deadline, the already printed texts, corrected by 20 June 1959, have been completely recast to enable the standard of presentation to conform to the traditions of the Oldenbourg Verlag. The material has been rearranged into seven chapters on considerations of logic, without any attention to the chronological order of the meetings. It includes approximately 60 papers and short reports on 12 symposia. All papers are in English or French, preceded by abstracts in these languages, also German, Russian and Spanish. There are English and French Indexes and the list of participants brings the total to 520 pages. The Preface has been written by the Director-General of UNESCO.

The volume may be ordered from booksellers at prices alternatively quoted on the dust jacket as U.S. $25.00; £7 7s. 0d.; N.F. 100.00; D.M. 84.00.

The printers and publishers are to be congratulated on making this volume available within twelve months of the Conference.