What EVERYBODY should know about ALGOL

By B. Higman

PART I: "O" LEVEL

if *you never use mathematics* **then** *stop reading right now* **otherwise if** *you dont want to use ALGOL* **then** *retire on your pension* **otherwise** *study this note carefully*

That was a good and valid ALGOL statement. Orthodoxy prefers the shorter **else** to the longer **otherwise,** but these bold type words (they would be underlined if hand- or typewritten) are only a trick to get round the paucity of symbols on the typewriter, and to argue about that is like arguing about whether, if you haven't a multiplication sign on your typewriter, you type "a **times** b" or "a **multiplied by** b." Both of them are clumsy substitutes for the cross. (A letter X is an ambiguous substitute, which will not do at all.)

ALGOL has come into existence because ordinary mathematical notation is inadequate to describe the processes of numerical calculation, and needs extending for this purpose. ALGOL is nothing more than the extra symbols required to describe numerical calculations and the rules about how to use them. Its alleged difficulties are greatly exaggerated. We all know that

$$a + b \times c + d$$

has a trap concealed in it; the $b \times c$ must be worked out first. In ALGOL too, if you don't know the rules, there are traps you can fall into; they are neither worse nor more numerous than those associated with algebraic symbols.

Every ALGOL statement calls for action. This is quite unlike other mathematical statements, such as $(a + b)(a - b) = a^2 - b^2$, which merely tell you something.* There are three rules about the use of statements beginning with **if**.

(1) The general form is

if ⟨an assertion⟩ **then** ⟨action if assertion is true⟩
else ⟨action if assertion is false⟩.

(2) If the second action is "do nothing" then it and the **else** can both be left out altogether.

(3) The second action, but not the first, can be another **if** statement.

This third rule prevents confusion about which **else** belongs to which **if,** and we shall see later that if we want to get around it, we can do so by putting brackets around the first action to remove the confusion. Notice that

$$\textbf{if } ax + b = 0 \textbf{ then } x = -b/a$$

is not a valid ALGOL statement because the **then** is followed by an assertion, not by an action.

Actions can be built up out of simple ones and then given names. That is how our opening sentence can be a valid statement; if it occurred in a program then the three actions it involves would have been defined in detail previously. Ninety-nine per cent of simple actions are of one of two sorts: giving a variable a value, and jumping (as in "go back to square one"). Successive simple actions are separated by semicolons. Semicolons also separate definitions (which do not qualify to be called actions).

Now it is a dreadful thing, but mathematical notation itself is not as unambiguous as it ought to be. Let l, m, n be three integers. Consider

$$ln(m^2 + m + 1).$$

Do *you* know whether this is $l \times n \times (m^2 + m + 1)$ or the natural logarithm of $m^2 + m + 1$? One thing is certain, it is asking a lot to expect a machine to decide such a question in the way you would, from the context. And ALGOL is meant to be "understanded of the machines" as well as "of the people." So, to settle this once for all, there is a rule about names. Any sequence of letters, or of letters and digits beginning with a letter, counts as one name. Furthermore, spaces, carriage returns, etc., are all to be ignored. So "BLACK SMITH" and "BLACKSMITH" count as the same name. (But capital and small letters count as different letters if you have the means to distinguish them.) This settles the dispute over "*ln*" in favour of the natural logarithm. It also means that we must take care to write

not	$2ab$	but	$2 \times a \times b$
not	$cos\ x$	but	$cos(x)$

but we have as wide a choice of names as we can ever need:

A, *B*, *C*,
alpha, *beta*, *gamma*,
aleph, *beth*, *gimel*,
*OC*45, *V*1*a*, *Minor*1000,
Temperaturpotenzreiheentwicklung,
Calc of moments,
etc. etc.

* But ALGOL has enough of the spirit of Pure Mathematics that it considers "do nothing" to be one possible action. And do not be misled here, "do nothing" is not "stop," it is more like "miss out on this turn."

Every freedom has its associated responsibility, however, and in this case, before we are allowed to use any name, we are required to define it. (An exception can be made in practice of the universally understood *sin*, *cos*, etc.) For the moment we can consider three ways of doing this:

(1) by listing those names which stand for real variables or integral variables with an appropriate heading to the list;
(2) by attaching the name, as a label, to a statement; and
(3) by giving details of any elaborate action to which we wish to give a name.

ALGOL adds the imperative mood to algebra. In algebra we assert that x is equal to y by writing "$x = y$." In ALGOL we need a symbolic representation of "make x equal to y," and for this "$x := y$" has been chosen, the colon and the equals together forming a single symbol (like "th" being equivalent to θ). A second imperative is required for jumps; this is the (single) symbol **go to.**

We can now write a program! At least, if we take one obvious liberty, we can. The following program will print the numbers 1, 4, 9, 16, 25 . . .

```
integer a; a := 0; R: a := a + 1; print (a × a); go
                     to R;
```

It is a trivial program, of course, but it does show the first two sorts of definition and both sorts of action. a is a variable, defined by a list, and R is a label. defined by prefixing it to a statement with a colon between. (The liberty is to assume that *print* does not have to be defined. Actually any detailed definition of it will have to involve not only ALGOL but also some knowledge of machine hardware; this must be true of all input and output, and ALGOL allows, without specifying details, the definition of an elaborate action to be in machine code in such cases.)

Solving a quadratic is "O" level algebra. It is safe to say that no one who has read as far as this would be daunted by it. But it also happens to be a nasty job to program on account of the variety of special treatments needed for special cases and to guarantee retaining full accuracy. So it will serve as a very suitable example of a "real" program after we have looked at one or two further points.

(1) Suppose we wanted to write

```
if a = 0  then  p := 0; q := 1
          else  p := 1; q := 2;
```

If the intention is what the layout suggests, it will be misinterpreted, because as soon as the first semicolon is reached, it will be assumed that this is the end of a "short" **if** statement (in which there is no **else** because the second action is "do nothing"). We need some form of brackets to enclose the "$p := 0;\ q := 1$," such as

```
if a = 0  then {p := 0; q := 1}
          else {p := 1; q := 2};
```

and these ought for convenience to be different from the round brackets used in algebraic expressions. ALGOL uses **begin** and **end** for these brackets, and writes

```
if a = 0 then begin p := 0; q := 1 end else begin
   p := 1; q := 2 end;
```

(2) These brackets are used in the definition of an elaborate action. To form such a definition we write out the details, surround them by these brackets, and precede the result by "**procedure** ⟨the name⟩;". Though not the whole story, this is the gist of defining the names of elaborate actions. It is also usual to surround the whole program by **begin** . . . **end.**

(3) Definitions must follow immediately after a **begin.** If this is the one at the start of the program, then the names are defined for the whole program, but by putting them after some other **begin** they can be made local to a part of the program.

(4) It so happens that only three symbols can legally follow **end**, namely, another **end**, a semicolon, or an **else**. So the rules allow commentary which does not include any of these three symbols to be inserted after an **end.** This is particularly helpful in allowing us to insert something after an **end** to help the human reader to identify which **begin** it belongs to. Such commentary is ignored by a machine.

(5) Lastly, we need two more input/output facilities; *next* is short for "next number on the input tape," and *printtext*(⌜X ⌝) prints, not the value of X, but the actual letter X. And in case *sign*(x) is not universally understood, it is $+1$ for positive x, -1 for negative x, and 0 for zero x.

Here, then, is a program for accepting the three coefficients of a quadratic equation (that of x^2 first) and printing out the roots. The use of the usual formula to obtain the numerically greatest root, followed by use of the knowledge of the product of the roots to obtain the smaller, is a well-known technique for avoiding loss of significant figures in the latter.

```
begin real a, b, c;
     a := next; b := next; c := next;
     if a = 0 then begin
          if b = 0 then begin
               if c = 0 then printtext (⌜indeterminate⌝)
                    else printtext (⌜both infinite⌝);
               go to nineteenth hole end b = 0
          else print (−c/b); printtext (⌜and infinite⌝)
            end a = 0

     else begin real d;
          d := b × b − 4 × a × c;
          if d < 0 then go to complex;
          d := sqrt(d);
          if b ≠ 0 then d := −b − d × sign(b);
          print (d/(2 × a));
          if c = 0 then print (0) else print (2 × c/d);
          go to nineteenth hole;

complex: print (−b/(2 × a));
          printtext (⌜+ ␣ and ␣ − ␣ i ␣ ×⌝);
          print (sqrt(−d)/(2 × a))   end a ≠ 0;

 nineteenth hole: end program
```

Note that the action labelled *nineteenth hole* is a "do nothing" action, to be jumped to when the program ends in the middle, as it were. The name has been chosen to recall the fact that the program is finished before this statement is reached. *d* is a "memo-pad" variable defined only for the second part of the problem. Since spaces are ignored, *printtext* has a special convention for spaces.

PART II: "A" LEVEL

Can you read the following formulae aloud, in such a way that it is possible for another person to take them down, and the correctness of the result to be used in evidence?

(1) $e^{p+q} + r$
(2) $F_G f_g + F_g f_G + F_{G-g} f_{G+g} + F_{G+g} f_{G-g}$
(3) $x^{2}_{m_1, m_2^2} + x^{-2}_{m_1^2, m_2}$

Modern mathematics is a game for hand and eye, both of which operate two-dimensionally, rather than for mouth and ear, which are limited to one dimension (unless you are willing to raise and lower the pitch of your voice for exponents and subscripts, and even then example (3) would be a musical *tour de force*).

Now paper tape is one-dimensional, in this sense. So ALGOL has inevitably something of the clumsiness of spoken mathematics, how much depending on the installation. In ALGOL reference language these three formulae become:

(1) $e \uparrow (p + q) + r$
(2) $F[G] \times f[g] + F[g] \times f[G] + F[G - g] \times f[G + g] + F[G + g] \times f[G - g]$
(3) $x[m[1], m[2] \uparrow 2] \uparrow 2 + x[m[1] \uparrow 2, \mathrm{m}[2]] \uparrow (-2)$

In paper tape with less than seven holes there will be shift symbols as well, corresponding to such words as "capital" and "small" as inserted when speaking.

The upward pointing arrow raises the following number, name, or bracketed expression to exponent level. A functional name such as *sin* obviously includes the following bracket containing its argument. A number must be unsigned or else bracketed. Square brackets lower their contents to subscript level and may be nested as often as occasion demands. Except when actually feeding a program into a machine, the more ordinary conventions may be used if preferred.

Subscripted variables are defined by an extension of the list method, incorporating information as to the range of the subscripts, thus

array X, $Y[1 : 3]$, $Z[0 : 2]$, A, $B[1 : 3, 1 : 3]$;

defines X, Y and Z as three-component vectors (Z having a different range of suffixes from X and Y) and A and B as 3×3 matrices.

Only algebraic suffixes, i.e. suffixes which are themselves variables or expressions, are treated in this way. If y_i and y_o stand for input and output values of y, they would be simply *yi* and *yo* in ALGOL. For if i happens to have the value 3 at any moment, then $y[i]$ is $y[3]$, but *yi* is not *y*3.

We can cook up far worse formulae than (1)–(3) above! How about a determinant whose components are definite integrals whose limits are themselves definite integrals? Fortunately this sort of thing can always be steam-rollered out by means of functional notation. For example, one can write

$$\sum_{h=a}^{b} f(h) \quad \text{as} \quad sigma(f(h), h, a, b)$$

This makes it just too easy. And the catch? Only that now we have to supply a definition for *sigma*. But that is, in itself, enough to call for a new section.

Of course we don't work out a definition of *sigma* if we can find one in the library. The library probably gives us something like

```
real procedure sigma(w, x, y, z); value y,z; real w;
integer x, y, z;
  begin real s; s := 0;
    for x := y step 1 until z do s := s + w;
    sigma := s end
```

This is not as bad as it looks, if we take its features one by one.

(1) It is a more elaborate form of the structure
procedure ⟨name⟩; **begin** . . . **end**
which we mentioned in Part I.

(2) The initial **real** means that it is not so much "an elaborate action" as a function with real values needing "elaborate action" to work it out. The function is assigned a value in the final statement *sigma* := *s*.

(3) When the program meets *sigma*($f(h)$, h, a, b) it will carry out the actions described in the definition, with $f(h)$ for w, h for x, a for y and b for z.

(4) The process may break down if the quantities replacing x, y and z are not integral, or the quantity replacing w is not real (which in this context includes integral).

(5) The list headed by the symbol **value** means that the quantities replacing y and z are determined

once for all at the start. By contrast, those replacing w and x do so algebraically, so that every time x occurs in the definition h is referred to, and every time w occurs, $f(h)$ is worked out using the current value of h.

(6) The new symbols, **for, step, until, do** can be taken at their face value. As with **then,** the effect of a **do** lasts until the next semicolon which is not protected by statement brackets **begin** . . . **end.**

Which should be crystal clear! If it isn't, try reading it a second time (a process called iteration), though as some of the above comments can be broadened to cover a wider variety of contexts, reading on may also help. For example, there are other types of statement beginning with **for.** Thus to evaluate

$$\sum_{x=1,2,3} w[x]$$

we can write

$s := 0$; **for** $x := 1, 2, 3$ **do** $s := s + w[x]$;

and to evaluate

$$\sum_{\substack{x=1\ldots 2k \\ \text{except } k\pm 1}} w$$

we might write

$s := 0$;
for $x := 1$ **step** 1 **until** $k - 2, k, k + 2$ **step** 1 **until** $2 \times k$ **do** $s := s + w$;

and there is also a third option provided which would deal with, say,

$$\sum_{n=1}^{\infty} \frac{1}{n^p} \text{ to 9 significant figures,}$$

namely,

$s := 0$; **for** $n := 1, n + 1$ **while** $n^p < 10^{10}$ **do** $s := s + n^{(-p)}$;

The effect of the list headed by **value** is quite subtle, but may broadly be described under three heads. (1) The procedure makes its own copy of each quantity in this list to do what it likes with, and so protects those copies of these variables which are the property of the main program. In this it is like a man who makes his own copy of a crossword puzzle so as not to spoil the original for other members of the family. (2) If the procedure is dealing with something like $sigma(k^2, k, 1, p^2 - pq + q^2)$, then if the z were not in the value list, the procedure would be in duty bound to re-evaluate $p^2 - pq + q^2$ every time it checked x against it, just in case p or q had somehow got altered in the meanwhile, which would be a great waste of time. But, (3), it is the very fact that w is *not* on the value list, and *is* different every time it is worked out, that makes the procedure work at all.

In this way nothing is barred that conforms to standard functional notation. Thus

$$\sum_{j=1}^{m} \sum_{k=1}^{n} f(j, k)$$

becomes

$$sigma\ (sigma(f(j, k), k, 1, n), j, 1, m)$$

Integrals can be dealt with in the same way, with this proviso, that in this case the definition, since it provides details, will presume a particular choice of quadrature rule (e.g. Simpson's). A word of warning is also necessary about the dummy variables in such expressions—j and k in the above example. These must be defined in the main program, but they are used by the procedure, and this makes it perilous to use them in the main program, except for very temporary purposes. If this warning is disregarded, then the procedure can throw a spanner into the works by surreptitiously altering the data behind the back of the main program.

PART III: "S" LEVEL

S is for Strings, Side effects, Switches, Synonyms, and Several other things that have not fitted into what has gone before.

Strings we have met before without giving them a name. A string is anything in inverted commas. The two output routines we used in solving a quadratic would have definitions beginning as follows:

procedure *print*(X); **real** X; . . .

and

procedure *printtext*(X); **string** X; . . .

ALGOL includes strings for obvious reasons, but it doesn't really like them.

Side effects we have also met without giving them a name, and as the enemy have tried to make a bogy of them, we think this bogy should be laid. The fact that $sigma(w, x, y, z)$ makes its own use of whatever main-program variable is substituted for x, and thus interferes with any long-term use of this variable, is an example of a side effect. This is fairly innocuous because the variable affected is visible among the parameters when *sigma* is called. But now suppose that we are curious to know how often *sigma* is actually called in a certain program. Then we add to the program:

(1) in its initial definitions an extra integer, *count*,
(2) among its initial orders, *count* := 0,
(3) inside *sigma*, *count* := *count* + 1, and
(4) at the end of the program, *print*(*count*).

And why not? We now have a side effect which is completely concealed when we are CALLING *sigma*, but then the whole idea was to make it automatic. It is true that this is the thin end of the wedge, and that we can now introduce the most dreadful concealed side effects just for the hell of it (see Ref. 13), but the plain fact of the matter is (1) that side effects as a principle are necessary, and (2) programmers who are irresponsible enough to introduce side effects unnecessarily will soon lose the confidence of their colleagues, and rightly so.

Switches are moment-of-truth devices. Suppose we have a big program which needs to solve a quadratic at several points. Then our earlier program will have to be converted into a procedure. Our reaction to the awkward cases may well be different at each of the several points. In these circumstances we could use a switch in the following way: we define *solve* thus:

```
procedure solve (a, b, c, x1, x2, S); value a, b, c;
  real a, b, c, x1, x2; switch S;
  begin integer n;
    if a = 0 then begin
      if b = 0 then begin
        if c = 0 then n := 1 else n := 2;
        go to moment of truth end
      else x1 := −c/b; n := 3 end
    else begin real d; d := b × b − 4 × a × c;
      if d < 0 then go to complex;
      n := 4; d := sqrt(d);
      if b ≠ 0 then d := −b − d × sign(b);
      x1 := d/(2×a);
surprise: x2 := if c = 0 then 0 else 2 × c/d;
      go to moment of truth;
complex: n := 5; x1 := −b/(2 × a);
      x2 := sqrt(−d)/(2 × a) end;
moment of truth: go to S [n] end
```

When we want to use this procedure, we shall write something like

solve (p, q, r, s, t, A);

say, and we must then include among the definitions one of the switch A, which we do in the form

switch A := *alpha*, *beta*, *gamma*, *delta*, *epsilon*;

where *alpha* . . . *epsilon* are the five labels (not necessarily all different, and not necessarily simple labels) showing where we wish to rejoin the main program in the various satisfactory or unsatisfactory situations represented by $n = 1, 2, 3, 4, 5$.

Among the Several other things, we may mention first that the **if** . . . **then** . . . **else** structure may be used not only for a conditional action, but also to state a conditional value. As an example, see the action labelled *surprise* in the above procedure, and compare it with the corresponding line in the "O" level program. Labels either after **go to** or in a switch may also take this form.

Secondly, an assertion, such as comes between **if** and **then,** has as its "value" either **true** or **false.** If we want to give a name to such an assertion, the name is listed as a logical, or **boolean** variable. Such values and variables imply the use of logical functions such as **and** (also written & or $\wedge$) and **or** (also written $\vee$). To show how they are used:

```
boolean x, y;
  x := the programme is bad;
  y := it is time for bed;
  if x or y then switch off the television;
```

An **and** or an **or** between two booleans is structurally similar to a **plus** or a **minus** (i.e. a + or −) between two numerical quantities, except that **and** is more like **times** since it takes priority over **or**. Similarly one can write, say, *firstentry* := **true** just as one can write $x := 25$.

S is for Synonyms because it is hoped that an exposition of this type gains by the avoidance of technical terms. Nevertheless, it is as well to have some sort of glossary, so here is a list of those terms we have used which have technical equivalents:

for name	read identifier
action	statement
definition	declaration (except those defining formal parameters of procedures, when "specification" is used)
elaborate action	procedure ("identifier" or "declaration" according to context)
assertion (when not a simple name)	boolean expression
algebraic replacement	call by name (not by value)

Further, a **begin** . . . **end** is called a "compound statement" if no definitions follow the **begin**, or a "block" if there are definitions. The distinction is important, because of the rule that names must be defined before they are used, which means that a block must be entered through its **begin**, whereas there is no objection to jumping to a label in the middle of a mere compound statement.

PART IV: SOME QUESTIONS ANSWERED

Q. How many more of these underlined words are there?

A. Only three, **comment, label** and **own.** The first seemed too obvious to need explanation, the second is only used as a list heading, and the third looks so unlikely to be used very much that it is safe to forget it. There are also several mathematical and logical signs such as $\neg$ (or **not**) and $\div$ (for integral division) which we have not mentioned.

Q. Is the use of English words in this way internationally agreed?

A. Yes. But it cannot be too strongly emphasized that they are not words but symbols. To a German, it is not so much a question of learning that **if** is "wenn," as that it must be followed somewhere by its **then** just as an integral sign must be followed somewhere by its $d(\;)$. And, conversely, for a machine to accept **wenn** instead of **if**, is like our accepting that the written and printed forms of the same letter are equivalent, and for purely local purposes would be a trivial variant.

Q. ALGOL 58 was followed by ALGOL 60. How long can we trust ALGOL 60 to remain?

Q. Is ALGOL good enough—is it not too early to freeze its form?

A. These questions are complementary. ALGOL 58 was very tentative, but ALGOL 60 is altogether different. The answer to both these questions is that as far as one can tell ALGOL 60 is likely to be added to but not changed. (Possible additions are permission to define a variable to be **complex**, and some technique for actually manipulating strings.)

Q. Won't it be a long time before ALGOL programs can be accepted for direct input to a machine?

A. ALGOL programs are already accepted at the Mathematical Centres at Amsterdam, Copenhagen, Stockholm and Mainz for direct input, and two English firms have ALGOL facilities in the "service trials" stage for their most up-to-date machines (the English Electric KDF9 and the Elliott 503 or 8,000-word 803). In this connection it is worth remarking that a phrase like KDF9 ALGOL refers to the input conventions used, methods of handling library procedures, and so on, which ALGOL leaves open; it does not imply any fundamental differences from official ALGOL.

Q. Has ALGOL any real advantages over, say, Mercury Autocode?

A. Once upon a time a nuclear physicist wanted the value of

$$\int_0^1 (\ln u)^{-6} \int_0^{e^{-k/(-\ln u)^{\frac{1}{2}}}} (-\ln w)^{-n}\, dw\, du$$

for certain integral values of n and k. He had no experience of computer programming in the ordinary sense, but he had learned to use ALGOL. He also knew enough to avoid asking any machine to evaluate ln (0), and therefore put in a certain amount of thought as to how to get around the problems posed by his limits. He found a library procedure for evaluating an integral by Simpson's rule, and this he copied in among his definitions, noting that in addition to the obvious requirements (limits, etc.) it also required him to specify the accuracy required, which he thought was reasonable, and a curious quantity, V, which he thought the machine might well have supplied for itself. He also noticed that the library procedure used some of the same symbols which he was proposing to use, but this did not worry him, because he knew that the ALGOL concept of "local definitions" would take care of this automatically. He then completed his program in the form given below, handed it to the machine operators together with a list of values of k and n, and very soon he received back the results he required.

```
begin real u, w; integer k, n;
      real procedure Simps (F, x, a, b, delta, V);
          value a, b, delta, V; real F, x, a, b, delta, V;
          comment integrates F with respect to x from a
            to b by Simpson's rule, halving the mesh size
            until the proportionate difference between
            two successive approximations is less than
            delta.  V must be supplied as anything
            greater than three times the largest value of
            abs(F) in the interval;
          begin integer n, k; real h, J, I;
            V := (b − a) × V; n := 1; h := (b − a)/2;
            x := a; J := F; x := b; J := (J + F) × h;
          J1: b := 0;
          for k := 1 step 1 until n do begin
              x := (2 × k − 1) × h + a;   b := b + F
                end;
          I := 4 × h × b + J;
          if abs(I − V) > abs(V) × delta  then  begin
            V := I; J := (I + J)/4; n := 2 × n;
            h := h/2; go to J1 end;
          Simps := I/3 end Simps;
actual program: n := next; k := next;
      print(Simps(if 10⁻⁷ > u or u > 0·999999 then 0
         else (ln(u))↑(−6) ×
          Simps(if 10⁻⁷ > w then 0 else (−ln(w))↑(−n),
            w, 0, exp(−k/sqrt(−ln(u))), 0·001, 10³⁵),
            u, 0, 1, 0·001, 10³⁵));
      go to actual program   end
```

This story is substantially true. I received it second hand, and can not vouch for minor details such as how the various values of n and k were handled, but the basic *print* statement is as I received it.

I challenge any user of Autocode to produce anything comparable in either directness of relation to the original statement of the problem, or in clarity, supposing discussion should arise at a later date concerning exactly how the values were arrived at.

Bibliography

1. Naur, P. (Ed.) (1963). "Revised Report on the Algorithmic Language ALGOL 60," *The Computer Journal*, Vol. 5, p. 349.
2. Naur, P. (Ed.) (1960). "Report on the Algorithmic Language ALGOL 60," *Numerische Mathematik*, Bd. 2, S. 106–36.
3. Dijkstra, E. W. (1962). *A Primer of ALGOL 60 Programming*, London: Academic Press.

4. Higman, B., and Goodman, R. H. (1963). *The Language of Computing: A Programmed Introduction to ALGOL*, London: English Universities Press (in press).
5. McCracken, D. D. (1962). *A Guide to ALGOL Programming*, London and New York: Wiley.
6. Dijkstra, E. W. (1962). "Operating Experience with ALGOL 60," *The Computer Journal*, Vol. 5, p. 125.
7. Duncan, F. G. (1962). "Implementation of ALGOL 60 for the English Electric KDF9," *The Computer Journal*, Vol. 5, p. 130.
8. Duncan, F. G. (1963). "Input and Output for ALGOL 60 on KDF9," *The Computer Journal*, Vol. 5, p. 341.
9. Hoare, C. A. R. (1962). "Report on the Elliott ALGOL Translator," *The Computer Journal*, Vol. 5, p. 127.
10. Hoare, C. A. R. (1963). "The Elliott ALGOL Input/Output System," *The Computer Journal*, Vol. 5, p. 345.
11. Hockney, R. W. (1962). "ABS12 ALGOL: an Extension to ALGOL 60 for Industrial Use," *The Computer Journal*, Vol. 4, p. 292.
12. Woodger, M. (1960). "An Introduction to ALGOL 60," *The Computer Journal*, Vol. 3, p. 67.
13. Knuth, D. E., and Merner, J. N. (1961). "ALGOL 60 Confidential," *Communications of the A.C.M.*, Vol. 4, p. 268.

Correspondence

To the Editor,
The Computer Journal.

A Hardware Representation for ALGOL 60 *using Creed Teleprinter Equipment*

Sir,

In your January issue, you published a paper by J. M. Gerard and A. Sambles describing the ALGOL 60 hardware representation for 5-hole Ferranti coded Creed teleprinter equipment. The paper contains many references to the KDF 9 character set, and, in view of the authors' conclusions, could be mistakenly taken to be the representation adopted by KDF 9 users. This is in fact not the case; a Working Party set up to consider the matter by our KDF 9 Users Group has recently agreed a 5-hole representation for ALGOL 60 which differs from the published proposals in the following instances:

	REFERENCE LANGUAGE	KDF 9 FLEXOWRITER	GERARD AND SAMBLES	EECUA KDF 9 5-hole CREED
(1)	$\uparrow$	$\uparrow$	*POW	**
(2)	]	]	)	*)
(3)	⌜	$\underline{[}$	£	*Q
(4)	⌝	$\underline{]}$	?	*U
(5)	;	;	. ,	* ,
(6)	:=	:=	.=	*=

The reasons for the choice of the above 5-hole representation of ALGOL for use on KDF 9 were briefly as follows:

(1) ** was adopted in preference to *POW as it is desirable to have a non-alphabetic representation. It was, of course, suggested by the equivalent representation in FORTRAN.

(2) The use of) for] is directly opposed in philosophy to that adopted by E.E. Co. in writing their Compilers, namely that all basicsymbols must have a context-free representation. Further, the representation adopted removed an implied restriction on the use of] within strings.

(3) and (4) *Q and *U (quote and unquote) were adopted for string quotes ⌜ ⌝ as they are a more natural representation than £ and ?. Further, the use of ? (5-hole binary 29) is inconvenient on KDF 9 due to the hardware restriction that 5 channel binary 29 is the "End of message" character.

(5) and (6) *, and *= were adopted in preference to ., and .= on the grounds of consistency, and follow De Havillands' approach in implementing ALGOL on Pegasus.

Yours faithfully,

J. M. R. Watson (Chairman, Working Party on 5-hole Working)

G. M. Davis (Secretary, EECUA).

English Electric Computer Users Association,
London Computer Centre,
Queens House, Kingsway, W.C.2.
18 March 1963

To the Editor,
The Computer Journal.

Sir,

The arguments used by Gerard and Sambles (1963) to support their choice of a hardware representation for ALGOL 60 in terms of Ferranti Pegasus five-track paper-tape code are not ideally consistent or convincing, as the following comments indicate.

1. While $\rightarrow$ for : may be thought suggestive when used in array declarations, it can hardly be considered so when used as the separator between a label and a statement. The alternative .. would appear to be suitable (and perhaps suggestive ?) in both cases. The argument that this is too similar to the symbol ., would, of course, apply equally to the symbols : and ; .

2. It seems a pity that the symbol * * was not adopted for $\uparrow$, in conformity with FORTRAN and the Elliott 803 Telecode, instead of the clumsy *POW.

3. Since a single asterisk is elsewhere used to denote underlining of basic symbols, the choice of $* \geqslant$ and $* >$ to represent $<$ and $\leqslant$ respectively seems particularly unfortunate. If suggestiveness is a desideratum, why not $n \geqslant$ and $n >$?

4. If the above changes were made, $\rightarrow$ would remain as an escape symbol.

Yours faithfully,

G. H. L. Buxton.

91 Kingston Road,
Earlsdon, Coventry.
21 March 1963

Reference

Gerard and Sambles (1963). "A hardware representation for ALGOL 60 using Creed Teleprinter equipment," *The Computer Journal*, Vol. 5, p. 338.