THE REMAINING TROUBLESPOTS IN ALGOL 60*

D. E. KNUTH

This paper lists the ambiguities remaining in the language ALGOL 60, which have been noticed since the publication of the Revised ALGOL 60 Report in 1963.

There is little doubt that the programming language ALGOL 60 has had a great impact on many areas of computer science, and it seems fair to state that this language has been more carefully studied than any other programming language.

When ALGOL 60 was first published in 1960 [1], many new features were introduced into programming languages, primarily with respect to the generality of "procedures." It was quite difficult at first for anyone to grasp the full significance of each of the linguistic features with respect to other aspects of the language, and therefore people commonly would discover ALGOL 60 constructions they had never before realized were possible, each time they reread the Report. Such constructions often provided counterexamples to many of the usual techniques of compiler implementation, and in many cases it was possible to construct programs that could be interpreted in more than one way.

The most notable feature of the first ALGOL 60 Report was the new standard it set for language definition, based on an almost completely systematic use of syntactic rules that prescribed the structure of programs; this innovation made it possible to know exactly what the language ALGOL 60 was, to a much greater degree than had ever been achieved previously. Of course it was inevitable that a complex document such as the ALGOL 60 Report (roughly 75 typewritten pages, prepared by an international committee) would contain some ambiguities and contradictions, since it involves a very large number of highly interdependent elements. As time passed, and especially as ALGOL 60 translators were written, these problems were noticed by many people, and in 1962 a meeting of the international committee was called to help resolve these issues. The result was the Revised ALGOL 60 Report [2], which cleaned up many of the unclear points.

Now that several more years have gone by, it is reasonable to expect that ALGOL 60 is pretty well understood. A few points of ambiguity and contradiction still remain in the Revised ALGOL 60 Report, some of which were left unresolved at the 1962 meeting (primarily because of high feelings between people who had already implemented conflicting interpretations of ambiguous aspects), and some of which have come to light more recently.

In view of the widespread interest in ALGOL 60 it seems appropriate to have a list of all its remaining problem areas, or at least of those which are now known. This list will be useful as a guide to users of ALGOL 60, who may find it illuminating to explore some of the comparatively obscure parts of this language and who will want to know what ambiguous constructions should be avoided; and useful also to designers of programming languages, who will want to avoid making similar mistakes.

The following sections of this paper therefore enumerate the blemishes which remain. A preliminary list of all the known trouble spots was compiled by the author for use by the ALGOL subcommittee of the ACM Programming Languages committee in November 1963; and after receiving extensive assistance from the committee members, the author prepared a revised document which appeared in mimeographed form in *ALGOL Bulletin 19, AB19.3.7* (Mathematisch Centrum, Amsterdam, January 1965). The present paper is a fairly extensive modification of the *ALGOL Bulletin* article; it has been prepared at the request of several people who do not have ready access to the *ALGOL Bulletin* and who have pointed out the desirability of wider circulation.

The following list is actually more remarkable for its shortness than its length. A complete critique which goes to the same level of detail would be almost out of the question for other languages comparable to ALGOL 60, since the list would probably be a full order of magnitude longer.

This paper is divided into two parts, one which lists *ambiguities* and one which lists *corrections* which seem to be necessary to the Revised Report. The word "ambiugous" is itself quite ambiguous, and it is used here in the following sense: An aspect of ALGOL 60 is said to be ambiguous if, on the basis of the Revised ALGOL 60 Report, it is

*Reprinted from *Comm ACM* 10, 10, 1967, 611-617.

The preparation of this paper has been supported in part by the National Science Foundation and in part by the Burroughs Corporation.

possible to write an ALGOL 60 program for which this feature can be interpreted in two ways leading to different computations in the program, and if it is impossible to prove conclusively from the Revised Report that either of these conflicting interpretations is incorrect. So-called "syntactic ambiguities" are not necessarily ambiguities of the language in this sense, although the original ALGOL 60 Report contained some syntactic ambiguities that did lead to discrepancies. (See [3] and [4] for a discussion of the former ambiguities; and see correction 7 below and the discussion at the end of Section 3 in [5] for comments on syntactic ambiguities remaining in the Revised Report.)

The distinction between an "ambiguity" in ALGOL 60 and a "correction" that is necessary to the Report is not clear cut; for when the Report contains an error or contradictory statement, this might lead to ambiguous interpretations, and conversely any ambiguity might be considered an error in the Report. The difference is mainly a matter of degree; the true meanings of the points that merely need to be corrected are almost universally agreed upon by people who have studied the Report carefully, because of the overall spirit of the language in spite of the fact that some of the rules are incorrectly stated.

Frequent references are made in the discussions below to the numbered sections of the Revised Report [2], and the reader is advised to have this document available for comparison if he is going to understand the significance of the comments which follow.

People who have studied the ALGOL Report carefully have often been called "ALGOL theologians," because of the analogy between the Bible and the Revised Report (which is the ultimate source of wisdom about ALGOL 60). Using the same analogy, it is possible to view the following sections as a more or less objective discussion of the conflicting doctrines that have been based on these Scriptures.

1. Ambiguities

AMBIGUITY 1: SIDE EFFECTS

A "side effect" is conventionally regarded as a change (invoked by a function designator) in the state of some quantities which are **own** variables or which are not local to the function designator. In other words, when a procedure is being called in the midst of some expression, it has side effects if in addition to computing a value it does input or output or changes the value of some variable that is not internal to the procedure.

For example, let us consider the following program:

```
begin integer a;
  integer procedure f(x,y); value y,x; integer y,x;
    a := f := x + 1;
  integer procedure g(x); integer x; x := g := a + 2;
    a := 0; outreal (1, a + f(a, g(a))/g(a)) end.
```

Here both f and g have as a side effect the alteration of an external variable.

It is clear that the value output by this program depends heavily upon the order of computation. Many compilers find more efficient object programs are obtained if the denominator of a complicated fraction is evaluated before the numerator; if we first compute $g(a)$, then $f(a, g(a))$, then $a + f(a, g(a))/g(a)$, and if the evaluation of the **value** parameters in $f(a, g(a))$ is done in the order a, $g(a)$, then we get the answer $4\frac{1}{2}$. Other possible answers are $\frac{1}{3}$, $\frac{3}{5}$, $\frac{3}{2}$, $\frac{5}{2}$, $\frac{4}{3}$, $3\frac{3}{5}$, $3\frac{1}{3}$, $5\frac{3}{5}$, $3\frac{1}{2}$, and $7\frac{1}{2}$.

The major point left unresolved in the Revised Report was the ambiguity about side effects: Are they allowed in ALGOL 60 programs, and if so what do the programs mean? If side effects are allowable, then the order of computation must be specified in the following places: evaluating the primaries of an expression; evaluating the subscripts of a variable; evaluating bound pairs in a block head; evaluating **value** parameters; and (perhaps) the step-until element of a for clause. Note, for example, that **value** parameters (which are to be evaluated just after entry to a procedure, Section 4.7.3.1.) might conceivably be evaluated in the order of their appearance in the parameter list or in the value part.

An argument may actually be made for the opinion that side effects are implicitly outlawed by the fifth paragraph in Section 4.5.3.2; or at least that paragraph says that all side effects occurring during the evaluation of an if clause of a conditional statement must be cancelled if the Boolean expression comes out false! A similar situation occurs in Section 4.3.5 where side effects, occurring during the evaluation of a designational expression which is ultimately undefined, must presumably be nullified. (On the other hand, the wording of these sections is probably just an oversight, and the implication about side effects was probably not intended.)

How close does the Report come to prescribing the order of computation? Section 3.3.5 says "the sequence of operations within one expression is generally from left to right," but the context here refers to the order of carrying out arithmetic operations; and it does not say whether the value of the first term "a" in the above example should be calculated before or after the second term "$f(a, g(a))/g(a)$" since no "operation" in the sense of the Report is involved here. Section 4.2.3.1 says subscript expressions in the left part variables of an assignment statement are evaluated "in sequence from left to right." (So in the assignment statement

$$A[a + B[f(a)] + g(a)] := C[a] := 0$$

we are perhaps to evaluate "$a + B[f(a)] + g(a)$" first, then "$f(a)$" again, then "a"?)

Section 1, footnote 4, says "Whenever ... the outcome of a certain process is left undefined ... a program only fully defines a computational process if accompanying information specifies ... the course of action to be taken in all such cases as may occur during the execution of the computation." In Section 3.3.6 we read, "It is indeed understood that different hardware representations may evaluate arithmetic expressions differently." The latter remark was made with reference to arithmetic on **real** quantities (i.e., floating-point arithmetic), but it is remarkable when viewed also from the standpoint of side effects! Footnote 4 says essentially that ALGOL 60 is not

intended to be free of ambiguity, and much can be said for the desirability of incompletely specified formalisms; indeed, this incompleteness is the basis of the axiomatic method in mathematics and it is also the basis of many good jokes. But it is doubtful whether the ambiguity of side effects is a desirable one; for further remarks in this vein see Ambiguity 9.

In view of the ambiguities of side effects, which many people do not realize because they know only the interpretations given by the ALGOL compiler they use, the author has founded SPASEPA, the Society for the Prevention of the Appearance of Side Effects in Published Algorithms. Members and/or donations are earnestly solicited.

It may be of value to digress for a moment here and to ask whether side effects are desirable or not; should ALGOL or comparable languages allow side effects? Do side effects serve any useful purpose or are they just peculiar constructions for programmers who like to be tricky? Objections to side effects have often been voiced, and the most succinct formulation is perhaps that due to Samelson and Bauer in *ALGOL Bulletin 12*, pp. 7–8. The principal points raised are that (i) an explanation of the "use" of side effects tends to waste inordinate amounts of classroom time when explaining ALGOL, giving an erroneous impression of the spirit of the language; (ii) familiar identities such as $f(x) + x = x + f(x)$ are no longer valid, and this is an unnatural deviation from mathematical conventions; (iii) many "applications" of side effects are merely programming tricks making puzzles of programs; other uses can almost always be reprogrammed easily by changing a function designator to a procedure call statement. Essentially the same objections have been voiced with respect to the concept of parameters "called by name," which was the chief new feature of ALGOL 60.

Another objection to side effects is that they may cause apparently needless computation. For example, consider

"**if** $g(a) = 2 \vee g(a) = 3$ **then** 1 **else** 0"

in connection with the procedure $g(x)$ above; according to the rules of the Revised Report it is necessary to evaluate $g(a)$ twice, thereby increasing a by 4 even if the first relation involving $g(a)$ is found to be true.

We might also mention the fact that ALGOL's call-by-name feature is deficient in the following respect: It is impossible to write an ALGOL 60 procedure "*increment* (x)" which increases the value of the variable x by unity. In particular the procedure statement "*increment* $(A[i])$" should increase the current value of $A[i]$ by unity, where i is a function designator which may produce different values when it is invoked twice.

On the other hand, consider the following procedure:

```
real procedure SIGMA (i, l, u, x); value l, u; integer i, l, u;
    real x;
  begin real s; s := 0; for i := l step 1 until u do s := s + x;
    SIGMA := s end.
```

This procedure computes $\sum_{i=l}^{u} x$ and has the additional side effect of changing variable i. It is quite natural to be able to write

$$SIGMA\ (i, 1, m, SIGMA\ (j, 1, n, A[i, j])) \quad \text{for} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} A_{ij}$$

without adding special summation conventions to the language itself; this is a tame and unambiguous use of side effects which also is the principal example that has been put forward to point out the usefulness of parameters called by name. (See [6] for further discussion.)

If we alter the above procedure by inserting "**integer** $i0$; $i0 := i$;" after "**real** s;" and "; $i := i0$" before "**end**", we would find that no side effect is introduced as a consequence of the total execution of the function $SIGMA\ (i, l, u, x)$, provided the actual parameter x does not involve side effects. So the above example does not constitute an inherent use of side effects; in fact, a study of this particular case indicates that it might be better to have some sort of facility for defining dummy variables (like i and j) which have existence only during the evaluation of a function but which may appear within the arguments to that function.

We should remark also that the principal objection to allowing parameters called by name, even in natural situations like the above example, is that the machine language implementation of these constructions is necessarily much slower than we would expect a simple summation operation to be; the inner loop (incrementation of i, testing against u, adding x to s) involves a great deal of more or less irrelevant bookkeeping because i and x are called by name, even on machines like the Burroughs B5500 [12] whose hardware was specifically designed to facilitate ALGOL's call by name. The use of "macro" definition facilities to extend languages, instead of relying solely on procedures for this purpose, results in a more satisfactory running program.

Other situations for which function designators with side effects can be useful are not uncommon, e.g., in connection with a procedure for input or for random number generation. Side effects also arise naturally in connection with the manipulation of data structures, when a function changes the structure while it computes a value; for example, it is often useful to have a function "*pop*(S)" which deletes the top value from a "stack" S and which retains the deleted value as its result. See also [7] for examples of Boolean function designators with side effects that are specifically intended for use in constructions like $p \vee q$, where q is *never* to be evaluated when p is true and where p is to be evaluated first in any case.

The objection above that $x + f(x)$ should be equal to $f(x) + x$, because of age old mathematical conventions, is not very strong; there are simple and natural rules for sequencing operations of an expression so that a programmer knows what he is doing when he is using side effects. The people who complain about "$x + f(x)$" are generally compiler writers who don't want to generate extra code to save x in temporary storage before computing $f(x)$, since this is almost always unnecessary. Such inefficiency is the real reason for the objections to side effects. These same people would *not* like to see "$x = 0 \vee f(x) = 0$" be treated

the same as "$f(x) = 0 \vee x = 0$" since the former relation can be used to suppress the computation of "$f(x) = 0$" when it is known that "$x = 0$"; in fact one naturally likes to write "$x = 0 \vee f(x) = 0$" in situations where $f(0)$ is undefined.

AMBIGUITY 2: INCOMPLETE FUNCTIONS

The question of exit from a function designator via a **go to** statement is another lively issue. This might be regarded as a special case of point 1, since such an exit is a "side effect," and indeed the discussion under point 1 does apply here. Some further points are relevant to this case, however.

Some people feel this is an important feature because of "error exits." However, the same effect can be achieved by using a procedure call statement and adding an output parameter.

Two rather convincing arguments can be put forward to contend that this type of exit is not really allowed by ALGOL, so the matter is not really an ambiguity at all.

(a) In Section 3.2.3 we read, "Function designators define single numerical or logical values." An incomplete function would not. Or, if it would, there would be mysterious, ambiguous consequences such as this:

```
begin real x, y; real procedure F; begin F := 1;
    go to L end;
  x := F + 1; y := 1; L: end.
```

We question whether x is replaced by 2, and if so, whether y is replaced by 1 (thus incorporating simultaneity into the language?). After all, F rigorously defines the value 1 and "the value so assigned is used to continue the evaluation of the expression in which the function designator occurs." (Cf. Section 5.4.4.)

(b) The discussion of the control of the program in Section 5.4.3.2 is based entirely on the values of the Boolean expressions, and the language used there implicitly excludes such a possibility. In many places the Report speaks of expressions as if they have a value, and no mention is ever made of expressions that are left unevaluated due to exits from function designators.

A further point about incomplete functions (though not really part of the ambiguity) concerns the implementation problems caused when such an exit occurs during the evaluation of the bound expressions while array declarations are being processed. Since the control words for a storage allocation scheme are not entirely set up at this time, such exits have caused bugs in more than one ALGOL compiler!

AMBIGUITY 3: STEP-UNTIL

The exact sequence occurring during the evaluation of the "step-until" element of a for clause has been the subject of much (rather heated) debate. The construction

for V := A **step** B **until** C **do** S

(where V is a variable, A, B, C are expressions, and S is a procedure) can be replaced by a procedure call

for (V, A, B, C, S)

with suitable procedure called "*for*." The debate centers, more or less, on which of these parameters are to be thought of as called by value, and which as called by name.

Conservative ALGOL theologians follow the sequence given in Section 4.6.4.2 very literally, so that if statement S is executed n times, the value of A is computed once, B is computed $2\times n + 1$ times, C is computed $n + 1$ times, and (if V is subscripted) the subscripts of V are evaluated $3\times n + 2$ times. Liberal theologians take the expansion more figuratively, evaluating these things just once. There are many points of view between these two "extremist" positions. As a result, the following program will probably give at least four or five different output values when run on different present-day ALGOL implementations:

```
begin array V, A, C[1:1]; integer k;
  integer procedure i; begin i := 1; k := k + 1 end;
  k := 0; A[1] := 1; C[1] := 3;
  for V[i] := A[i] step A[i] until C[i] do;
  outreal (1, k) end;
```

The liberal interpretation gives an output of 4, the conservative interpretation gives something like 23, and intermediate interpretations give intermediate values; for example the compromise suggested in [9] gives the value 16.

The conservative argument is, "Read Section 4.6.4.2." The liberal arguments are: (a) "If Section 4.6.4.2 is to be taken literally, it gives a perfectly well defined value for the controlled variable upon exit. Since Section 4.6.5 says the value is undefined, however, it must mean Section 4.6.4.2 is not to be taken literally." (b) "The repeated phase *'the* controlled variable' is always used in the singular, implying that the subscript(s) of the variable need be evaluated only once during the entire for clause. Other interpretations make Section 4.6.5 meaningless."

Examination of published algorithms shows that in well over 99% of the uses of for statements, the value of the "step" B is $+1$, and in the vast majority of the exceptions the step is a constant. It is clear that programmers seldom feel the need to make use of any ambiguous cases. The liberal interpretation is clearly more efficient and it would be recommended for future programming languages; a programmer who really feels the need for some of the woollier uses of a for statement can be told to write the statements out by adding a tiny bit of program instead of using a for statement. Even though uses can be contrived for examples like

for x := .1 **step** x **until** 10^6

or

for y := 1 **step** 1 **until** $y + 1$

these are rewritten easily using the "while" element.

AMBIGUITY 4: SPECIFICATIONS

The wording of Section 5.4.5 can be interpreted as saying that parameters called by value must be specified only if the specification part is given at all! Furthermore, it is not stated to what extent, *if any*, the actual parameters must agree with a given specification, and to what extent

the specifications which do appear will affect the meaning of the program. For example, is the following program legitimate?

```
begin integer array A, B, C[0:10]; array D[0:10];
  procedure P(A, B, C); array A, B, C;
    begin integer i;
      for i := 0 step 1 until 10 do
        C[i] := A[i]/B[i]
    end;
  integer i;
  for i := 0 step 1 until 10 do
      begin A[i] := 1; B[i] := 2 end;
  P(A, B, C); P(A, B, D)
end.
```

If so, the assignment statement inside the procedure will have to *round* the result or not depending on the actual parameter used. Consider also the same procedure with the formal parameters specified to be *integer* arrays. For further discussion see [9].

AMBIGUITY 5: REPEATED PARAMETERS

Several published algorithms have a procedure heading like

procedure *invert* (*A*) *order*: (*n*) *output*: (*A*)

where two of the formal parameters have the same name. The Report does not specifically exclude this, and it does not say what interpretation is to be taken.

AMBIGUITY 6: VALUE LABELS

It has not been clear whether or not a designational expression can be called by value, and if so, whether its value may be "undefined" as used in Section 4.3.5. This may or may not be allowed by the language of Section 4.7.3.1 (which talks about "assignment" of values to the formal parameters in a "fictitious block"). Cf. Section 4.7.5.4; if a designational expression could be called by value, a switch identifier with a single component could be also, in the same way as an array identifier can be called by value. The first paragraph of Section 2.8 is relevant here also.

AMBIGUITY 7: OWN

This has so many interpretations it will take too much space to repeat the arguments here. See [8, 9] for a discussion of the two principal interpretations, "dynamic" and "static," each of which can be useful. The additional complications of own arrays with dynamically varying subscript bounds combined with recursion, adds further ambiguities; for one apparently reasonable way to define this, see [11].

AMBIGUITY 8: NUMERIC LABELS AND "QUANTITIES"

Most ALGOL compilers exclude implementation of numeric labels, primarily because a correct implementation requires an unsigned integer constant parameter to be denoted, in machine language, both as a number and a label. Consider for example

```
procedure P1(q, r); if q < 5 then go to r;
procedure P2(q, r); if r < 5 then go to q;
procedure W(Z); procedure Z; Z(2, 2);
··· W(P1); x := 0; W(P2); 2: ···
```

There is no ambiguity here in the sense we are considering, just a difficulty of implementation in view of the double meaning of a parameter "2."

The author has shown the following procedure to several authoritative people, however, and a 50% split developed between those saying it was or was not valid ALGOL:

```
procedure P(q); if q < 5 then go to q;
```

The idea of course is that we might later call $P(2)$ where 2 is a numeric label. Actually this seems to be specifically outlawed by Section 2.4.3 (the identifier q cannot refer to two different quantities). But consider

```
procedure P(q); if B(q) then G(q);
procedure G(q); go to q;
Boolean procedure B(q); B := q < 5;
```

Is this now valid?

Consider also

```
begin integer I; array A[0:0];
procedure P1(X); array X; X[0] := 0;
procedure P2(X); integer X; X := 0;
procedure call (X, Y); X(Y);
call (P1, A); call (P2, I) end
```

The identifier Y is used to denote two different quantities (an array and a simple variable) which have the same scope, yet this program seems to be valid in spite of the wording of Section 2.4.3.

The latter procedure is believed to be admissible because the expansion of procedure bodies should be considered from a dynamic (not static) point of view. For example consider

```
integer procedure factorial (n); integer n;
  factorial := if n > 0 then n × factorial (n − 1) else 1;
```

In this procedure body the call of *factorial* $(n - 1)$ should not be expanded unless $n > 0$, or else the expansion will never terminate. From the dynamic viewpoint the identifier Y in *call* (X, Y) never does in fact denote two different quantities at the same time.

Another strong argument can be put forward that even our earlier example "**if** $q < 5$ **then go to** q" is allowable. Notice that Section 2.4.3 does *not* say that an identifier may denote a string; but in fact a formal parameter *may* denote a string. Therefore we conclude that Section 2.4.3 does not apply specifically to formal parameters; this is consistent with the entire spirit of the Report, which does not speak of formal parameters except where it tells how they are to be replaced by actual parameters. The syntax equations in particular reflect this philosophy. Consider for example

```
procedure P(Q, S); procedure Q; string S; Q(S);
```

there is no way to use the syntax of ALGOL to show that "$Q(S)$" is a procedure statement and at the same time to reflect the fact that S is a string. We show S is an ⟨identifier⟩, but to show it is an ⟨actual parameter⟩ we must show it is either an ⟨expression⟩, an ⟨array identifier⟩, a ⟨switch identifier⟩, or a ⟨procedure identifier⟩, and it really is not any of these. So the only way to account for

this is to first replace Q and S by their actual parameters, in any invocation of P, and *then* to apply the syntax equations to the result.

The distinction between what is valid and what is not according to Section 3.4.3 is unclear.

AMBIGUITY 9: REAL ARITHMETIC

The precision of arithmetic on **real** quantities has intentionally been left ambiguous (see Section 3.3.6). In an interesting discussion van Wijngaarden [13] gives arguments to show among other things that because of this ambiguity it is not necessarily true that the relation "3.14 = 3.14" is the same as "**true**" in all implementations of ALGOL. As we have mentioned above, ambiguities as such are not necessarily undesirable; but is clear that ambiguities 1–8 are of a different nature than this one, since it can be quite useful to describe fixed ALGOL programs with varying arithmetic substituted.

So a language need not be unambiguous, but of course when intentionally ambiguous elements are introduced it is far better to state specifically what the ambiguities are, not merely to leave them undefined, lest too many people think they are writing unambiguous programs when they are not.

2. Corrections

CORRECTION 1: OMITTED **else**

In Section 3.3.3 it is stated that "the construction

else ⟨simple arithmetic expression⟩

is equivalent to the construction

else if true then ⟨simple arithmetic expression⟩

But the latter construction is erroneous since if fails to meet the syntax; we cannot write $A :=$ **if** B **then** C **else if true then** D. The original incorrect sentence adds nothing to the Report and means little or nothing to non-LISP programmers.

CORRECTION 2: CONDITIONAL STATEMENT SEQUENCE

In Section 4.5.3.2, the paragraph "If none ... dummy statement" should be deleted or at least accompanied by a qualification that it applies only to the second form of a conditional statement. This well-known error and also Correction 11 would have been fixed in the Revised Report except for the fact that these proposals were tied to other ones involving side effects; in the heated discussion which took place, the less controversial issues were overlooked.

The Revised Report changed the syntax of conditional statements and this makes Section 4.5.3.2. even more erroneous. And the explanation is incorrect in yet another respect, since control of the program should not pass to the statement called "*S4*" when the conditional statement is a procedure body or is preceded by a for clause.

Therefore Section 4.5.3.2 should be completely rewritten, perhaps as follows:

4.5.3.2. Conditional statement. According to the syntax, three forms of unlabelled conditional statements are possible. These may be illustrated as follows (with Section 4.5.4 eliminated):

if B **then** S_u
if B **then** S_u **else** S
if B **then** S_{for}

Here B is a Boolean expression, S_u is an unconditional statement, S is a statement, and S_{for} is a for statement.

The execution of a conditional statement may be described as follows: The Boolean expression B is evaluated. If its value is **true**, the statement S_u or S_{for} following "**then**" is executed. If its value is **false** and if the conditional statement has the second form, the statement S following "**else**" is executed. (This statement S may of course be another conditional statement, which is to be interpreted according to the same rule.)

If a go to statement refers to a label within S_u or S_{for}, the effect is the same as if the remainder of the conditional statement (namely "**if** B **then**," and in the second case also "**else** S") were not present.

CORRECTION 3: FOR EXAMPLE

The second example in Section 4.6.2 is not very good since (precluding side effects) it nearly always gets into an unending loop. Therefore, change "*V1*" to "*k*" in both places.

CORRECTION 4: FUNCTION VALUES

Two sentences of Section 5.4.4 should say " ... as a left part ... " rather than " ... in a left part ... " since a function designator may appear in a subscript. A clarification, stating that the value is lost if a **real**, **integer**, or **Boolean** procedure is called in a procedure statement, might also be added here.

Change sentence 2, Section 4.2.3 " ... a function designator of the same name ... ". This makes an implied rule explicit. Or else, consider

```
real procedure A;  A := B := 0;
integer procedure B(k);  if k > 0 then A else B := 2;
```

which appears to conform to all of the present rules.

CORRECTION 5: EXPRESSIONS

In the second sentence of Section 3, insert "labels, switch designators," after "function designators." This describes the constituents of expressions much more accurately.

CORRECTION 6: DIVISION BY ZERO

Insert after the second sentence of 3.3.4.2: "The operation is undefined if the factor has the value zero. In other cases," The present wording of this section seems to imply 1/0 is defined somehow.

CORRECTION 7: STRING SYNTAX

The advent of syntax-oriented compilers and the fact that the syntax of ALGOL is (in large measure) formally unambiguous, make it desirable to change the most flagrantly ambiguous syntax rule in the Report. Therefore it is suggested that in Section 2.6.1 the definition of open string be replaced by

⟨open string⟩ :: = ⟨proper string⟩ |
⟨open string⟩ ⟨string⟩ ⟨proper string⟩

CORRECTION 8: LIBRARY PROCEDURES

Section 2.4.3 says "[Identifiers] may be chosen freely (cf. however, Section 3.2.4, Standard Functions)." Section 3.2.4 says "Certain identifiers should be reserved for the standard functions of analysis, which will be expressed as procedures." If the quotation from 2.4.3 is not self-contradictory it seems to be saying that an identifier like "*abs*" may not be used by a programmer. But this would be disastrous since the list of reserved identifiers is not defined. A programmer using the name "*gamma*" for a variable may find out next year that this identifier is reserved for the gamma function. It should be made clear that any identifier may be redeclared (although this can of course lead to some difficulties when a procedure is copied from the literature into the middle of a program).

Moreover, the fourth paragraph of Section 5 specifically disallows the use of any procedures assumed to exist without declaration, except function designators denoting "standard functions of analysis." Thus, procedures such as "*inreal*," etc. for accomplishing input would have to be declared in any program which uses them! A suggested change (which I think most people would say was no change from the original intention) would be to drop the sentence "Apart from labels . . . must be declared" from the paragraph mentioned, and to add the following paragraph to Section 5:

"It is understood that certain identifiers may have meaning without explicit declaration, as if they were declared in a block external to the entire program (cf. Section 3.2.4). Such identifiers might include, for example, names of standard input and output procedures. Apart from labels, formal parameters of procedure declarations, and these standard identifiers, each identifier appearing in a program must be declared."

This paragraph makes available other types of identifiers if there is a need for them, e.g., an identifier denoting a real-time clock, or a label denoting a particular part of a control program, etc.

CORRECTION 9: OUTER BOUNDS

The statement of Section 5.2.4.2: "Consequently in the outermost block of a program only array declarations with constant bounds may be declared," should be amended to allow for the possibility of calls on standard functions (or other standard identifiers as noted in correction 8). The declaration

array $A[0: abs(-2)]$

is allowable in the outermost block, or the word "consequently" does not apply.

CORRECTION 10: LABELLED PROGRAMS

The syntax for ⟨program⟩ allows a program to be labelled but the remainder of the Report always talks about labels being local to some block. To rectify this, insert three words into Section 4.1.3:

" ... a procedure body or a program must be considered ... "

This is in fact the way a compiler should probably do the implementation (see [10]). As an example, consider the following:

A: **begin array** $B[1: read]$; *outarray* (B); **go to** A **end**;

CORRECTION 11: UNDEFINED **go to**

The use of the word "undefined" in Section 4.3.5 is highly ambiguous, and under some interpretations it leads to undecidable questions which would make ALGOL 60 truly impossible to implement. Under what conditions is a switch designator "undefined"? For example we could say it is undefined if its evaluation procedure makes use of real arithmetic, or if its evaluation procedure never terminates. By a suitable construction, the latter condition can be made equivalent to the problem of deciding whether or not a Turing Machine will ever stop.

The following procedure is an amusing (although unambiguous) example of the application of an undefined go to statement, which points out how difficult it can be for an optimizing ALGOL 60 translator to detect the fact that a procedure is being called recursively:

```
begin integer nn;
  switch A := B[1], B[2];
  switch B := A[G], A[2];
  integer procedure F(n, S); value n; integer n; switch S;
    begin nn := n; go to S[1]; F := nn end F;
  integer procedure G;
    begin integer n;
    n := nn; G := 0;
    nn := if n ≤ 1 then n else F(n−1, A) + F(n−2, A)
    end S;
  outreal (1, F(20, A)) end.
```

The output of this program should be 6765 (the twentieth Fibonacci number).

CORRECTION 12: CALL BY NAME

Instead of "Some important particular cases of this general rule" at the end of Section 4.7.5, it should be e.g., "Some important particular cases of this general rule, and some additional restrictions." The restrictions of Subsection 4.7.5.2 are not always special cases of the general rule, as shown in the following amusing example:

```
begin procedure S(x); x := 0;
  real procedure r; S(r);
  real x; x := 1;
  S (if x = 1 then r else x); outreal (1, x) end.
```

This program seems to have a historical claim of being the last "surprise" noticed by ALGOL punsters; it contains two unexpected twists, the first of which was suggested by P. Ingerman:

(a) Procedure r uses $S(r)$ to set the value of r to zero.

(b) The expansion of the procedure statement on the last line, according to the rules of "call by name," leads to a valid ALGOL program which has a completely different structure than the body of S:

if $x = 1$ **then** r **else** $x := 0$

Here an unconditional statement plus a conditional expression has become a conditional statement.

Fortunately both of these situations have been ruled out by Section 4.7.5.2.

Conclusion

For centuries astronomers have given the name ALGOL to a star which is also called Medusa's head. The author has tried to indicate every known blemish in [2]; and he hopes that nobody will ever scrutinize any of his own writings as meticulously as he and others have examined the ALGOL Report.

RECEIVED JANUARY 1967; REVISED JULY 1967

REFERENCES

1. NAUR, P. (Ed.) Report on the algorithmic language ALGOL 60. *Comm. ACM 3* (1960), 299–314.
2. NAUR, P., AND WOODGER, M. (Eds.) Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6* (1963), 1–20.
3. ABRAHAMS, P. W. A final solution to the dangling **else** of ALGOL 60 and related languages. *Comm. ACM 9* (1966), 679–682.
4. MERNER, J. N. Discussion question. *Comm. ACM 5* (1964), 71.
5. KNUTH, D. E. On the translation of languages from left to right. *Inf. Contr. 8* (1965), 607–639.
6. DIJKSTRA, E. W. Letter to the editor. *Comm. ACM 4* (1961), 502–503.
7. LEAVENWORTH, B. M. FORTRAN IV as a syntax language. *Comm. ACM 7* (1964), 72–80.
8. KNUTH, D. E., AND MERNER, J. N. ALGOL 60 confidential. *Comm. ACM 4* (1961), 268–272.
9. INGERMAN P. Z., AND MERNER, J. N. Suggestions on ALGOL 60 (Rome) issues. *Comm. ACM 6* (1963), 20–23.
10. RANDELL, B., AND RUSSELL, L. J. *ALGOL 60 Implementation*. Academic Press, London, 1964.
11. NAUR, P. Questionnaire. *ALGOL Bulletin 14*, Regnecentralen, Copenhagen, Denmark, 1962.
12. B5500 Information processing systems reference manual. Burroughs Corp., 1964.
13. VAN WIJNGAARDEN, A. Switching and programming. In H. Aiken and W. F. Main (EDS.), *Switching Theory in Space Technology*, Stanford U. Press, Stanford, 1963, pp. 275–283.