ISSN 0084-6198

# Algol Bulletin no. 38

DECEMBER 1974

CONTENTS



## Important notice to LIBRARIANS

If this copy of the ALGOL BULLETIN is to be placed in a library, please first detach pages 52-55 and put them with your copy of the "Revised Report on the Algorithmic Language ALGOL 68" which was sent to you as a Supplement to AB36 (these are in addition to the similar errata which you received with AB37). Better still, modify your copy in accordance with both sets of errata.

The ALGOL BULLETIN is produced under the auspices of the Working Group on ALGOL of the International Federation for Information Processing (IFIP WG2.1, Chairman Professor J.E.L. Peck, Vancouver).

The following statement appears here at the request of the Council of IFIP:

"The opinions and statements expressed by the contributors to this Bulletin do not necessarily reflect those of IFIP and IFIP undertakes no responsibility for any action which might arise from such statements. Except in the case of IFIP documents, which are clearly so designated, IFIP does not retain copyright authority on material published here. Permission to reproduce any contribution should be sought directly from the authors concerned. No reproduction may be made in part or in full of documents or working papers of the Working Group itself without permission in writing from IFIP".

Facilities for the reproduction and distribution of the Bulletin have been provided by Professor Dr. Ir. W.L. Van der Poel, Technische Hogeschool, Delft, The Netherlands.

The ALGOL BULLETIN is published approximately three times per year, at a subscription of $7 per three issues, payable in advance. Orders and remittances (made payable to IFIP) should be sent to the Editor. Payment may be made in any currency ( a list of acceptable approximations in the major currencies will be sent on request), but it is the responsibility of each sender to ensure that cheques etc. are endorsed, where necessary, to conform to the currency control requirements of his own country. Subscribers in countries from which the export of currency is absolutely forbidden are asked to contact the Editor, since it is not the policy of IFIP that any person should be completely debarred from receiving the ALGOL BULLETIN for such a reason.

The Editor of the ALGOL BULLETIN is:

Dr. C.H. Lindsey,
Department of Computer Science,
University of Manchester,
Manchester, M13 9PL,
England.

Back numbers, when available, will be sent at $3 each. However, it is regretted that only AB32, AB34, AB35, AB36 and AB37 are currently available. The Editor would be willing to arrange for a Xerox copy of any individual paper to be made for anyone who undertook to pay for the cost of Xeroxing.

## ALGOL 60

The Commentary on the ALGOL 60 Report, published in this issue, should come as a reminder that Working Group 2.1 is responsible for a whole family of languages, and not just for whatever may happen, at the time, to be its latest product. There is a very real intention that the changes proposed, if they should seem to be acceptable to the computing community, will be given official status.

We therefore need feedback, and to this end you will find a questionnaire on the last page. Although primarily intended for implementers, it may be filled in, so far as is applicable, by anyone with the interests of ALGOL 60 at heart. It is difficult for us, however, to ensure that all implementers are made aware of what is going on, and so we ask each one of you who uses the laguage to draw the attention of whoever implements your local version to this questionnaire, and to coerce him into filling it in. Never mind if this results in 500 separate people trying to coerce IBM, so long as it also catches that lone implementation in Timbuktu that nobody else knew about.

## ALGOL 68

The Revised Report is due to be published in Acta Informatica, Vol. 4, issues 2/3. The text will be as already issued by the University of Alberta as TR 74-3, as modified by ERRATA-2 (AB37.5), and as now further modified by ERRATA-3 contained in this issue. We apologise for the fact that there are so many changes. Most of them are quite trivial and do not affect the language defined, but nevertheless it is our aim to make the final document as near perfect as we can get it. Please elaborate them in your own copy.

Publication of the Revised Report does not imply that development of ALGOL 68 is now ended. The Working Group's Sub-committee on ALGOL 68 Support will be meeting in Boston in January and topics scheduled for discussion include ISO-code representations, independent compilation of program modules, partial parametrization, modals, etc.

## The ALGOL Bulletin

This issue of AB completes the first set of three isssues for which you have been asked to pay. We now have over 500 fully paid up subscribers. If you are one of those who have been with us since the start of the scheme, you will find your reminder notice enclosed. Please return it promptly to save unnecessary paperwork at this end. Regrettably, due to the increasing costs of paper and of postage, we have had to increase the price to $7 per three issues.

It is still my hope to publish three issues per year. That this has not proved possible during 1974 is principally due to lack of material, and the remedy for this is in your hands.

AB38.1.1 WG2.1 Future work

The following is the text of a resolution passed by WG2.1 at its meeting in Breukelen, Holland, in August 1974.

> According to its scope as contained in the bylaws of IFIP,
> "WG2.1 is responsible for:
> the continuing support of ALGOL 60;
> the promulagation and development of ALGOL 68;
> the exploration and evaluation of new ideas in the field of programming languages, possibly leading to further languages . . ."
> Whereas the Revision of ALGOL 68 is now complete, the pursuit of new ideas in the area of algorithmic languages becomes the primary concern of the group. To this end, WG2.1 strongly encourages contributions from a community which is wider than the current membership of the Working Group.

It is now the intention of the Working Group to "explore the concept space" in which new programming languages should lie, rather than to embark immediately upon the detailed specification of a new language. To this end, the next meeting of the Group, in late 1975, will take the form of an informal working conference at which papers will be presented and discussed. Anyone who feels that he has ideas to contribute is invited to contact the organiser, who is Steve Schuman, IBM Scientific Centre, Cedex 9, 92081 Paris La Défense, France.

AI 38.1.2 Conference on "Experience with ALGOL 68"

To be held at The Department of Computational and Statistical Science, The University of Liverpool, 8th to 10th April, 1975.

1. Background and Purpose Until recently, the limitation in the availability of ALGOL 68 to a few, mainly large, computer systems has inhibited the widespread acceptance of the language amongst computer users. This conference aims to review more recent attempts to make the language available on a wider variety of computers, including minicomputers, and to assess experience gained in teaching the language and in practical applications.

2. Scientific program The following list of topics suggests the primary accent of the conference. Invited and submitted papers on these topics will be presented. Submitted papers which depart from this program may be accepted if they are thought to be relevant to the general theme.

a) Algol 68 on minicomputers: the design of sublanguages, and implementation problems.
b) Algol 68 in a user environment: providing facilities for users of Algol 68.
c) Teaching Algol 68: teaching methods and problems encountered in introducing the language to both novice programmers and users of other languages.
d) Programming applications in Algol 68: The reaction of programmers.

3. Submission of Papers It is expected that about 12-20 papers will be presented, including some by special invitation. Panel discussions and workshop sessions may be arranged to allow for the presentation of less formal papers for which time cannot be allocated in the main program.

Participation in the conference does not require presentation of a paper, but all intending participants are invited to submit papers on relevant topics.

The following schedule has been established: Submission of title and abstract (500 words): 10th February, 1975; Notificaiton to authors of accepted papers: 1st March, 1975; Final version of paper due (2000-4000 words): 8th April, 1975. The proceedings of the conference will be published.

4. Your reaction In order to proceed with arrangements for the conference and to decide upon the final program, a preliminary indication of the likely response is required. If you are likely to be interested, please write at once to: Dr. P.G. Hibbard, The Department of Computational & Statistical Science, The University of Liverpool, Liverpool, L69 3BX.

## AB38.1.3 International Conference on ALGOL 68

June 10-12, 1975; Call for Papers. The 1975 ALGOL 68 Conference will be held at Oklahoma State University in Stillwater, Oklahoma. As in the past, this Conference is designed to provide a forum for discussion of implementation problems for ALGOL 68 and related languages. In addition, users are encouraged to attend and present their views at this Conference. Suggested topics for papers at this Conference include, but are not limited to: ALGOL 68 implementation, ALGOL 68 usage, Effects of ALGOL 68 on the design and/or implementation of other languages. Those wishing to submit a paper should send a working title to G.E. Hedrick by January 31, 1975, and send an abstract by April 30, 1975.

For further information, contact:

G.E. Hedrick,
Department of Computing and Information Sciences,
Oklahoma State University,
Stillwater, Oklahoma 74074. U.S.A.

R.M. De Morgan, I.D. Hill, B.A. Wichmann

A draft of this document was produced for a meeting of the IFIP Working Group 2.1 held in Breukelen, August 1974. Changes have been made as a result of comments received at that meeting.

The authors would like comments on whether the primitive IFIP based input-output system is worth including in this document. Comments would also be welcome on 5.2.4.3 which permits the declaration of arrays containing no element.

The authors have failed to reach agreement on whether subscripted controlled variables should continue to be allowed, or whether a restriction should be made (as in the IFIP subset) to allow only a variable identifier to be a controlled variable.

For the present this commentary has been written to make the restriction, although under 4.6.4.2 an explanation is given of how the operations on a subscripted controlled variable should be defined if allowed. If it is to be allowed, various consequential changes would be needed elsewhere in the document.

Comments on this issue would be welcomed.

Would AB readers please send comments to:

B. A. Wichmann, National Physical Laboratory,
Teddington, Middlesex, TW11 0LW U.K.

Owing to the limitations of the ISO-code printing device, the following representations are used:

| | |
|---|---|
| space | . |
| string quotes | ( ) |
| or | or |
| and | and |
| not | not |
| implies | impl |
| equivalent | equiv |
| not equals | ne |
| integer divide | div |
| ten | & |
| multiplication | * |

also syntactic brackets are not distinguished from less than and greater than.

# A commentary on the ALGOL 60 Revised Report

R.M. De Morgan, I.D. Hill, B.A.Wichmann

> "For, as on the one side common experience sheweth, that where a change hath been made of things advisedly established (no evident necessity so requiring) sundry inconveniences have thereupon ensued; and those many times more and greater than the evils, that were intended to be remedied by such change: So on the other side, the particular Forms .... being things in their own nature indifferent, and alterable, and so acknowledged; it is but reasonable, that upon weighty and important considerations, according to the various exigency of times and occasions, such changes and alterations should be made therein, as to those that are in place of Authority should from time to time seem either necessary or expedient ....
>
> And therefore of the sundry alterations proposed unto us, we have rejected all such as were either of dangerous consequence .... or else of no consequence at all, but utterly frivolous and vain ....
>
> Our general aim therefore in this undertaking was, not to gratify this or that party in any their unreasonable demands; but to do that, which to our best understandings we conceived might most tend to the preservation of Peace and Unity ....
>
> If any man, who shall desire a more particular account of the several Alterations .... shall take the pains to compare the present Book with the former; we doubt not but the reason of the change may easily appear."
>
> Preface to Book of Common Prayer 1662.

Over the past eleven years, various defects have been noted in the 'Revised Report on the Algorithmic Language ALGOL 60'. In general, these defects are of little consequence, but have resulted in unnecessary variations in the various implementations of ALGOL 60 thus impairing the portability of ALGOL 60 algorithms. The body responsible for ALGOL 60, Working Group 2.1 of the International Federation for Information Processing, therefore asked a small group under the chairmanship of C.A.R. Hoare to examine the maintenance of ALGOL 60. As a result of an appeal by Professor Hoare, about a dozen letters were received expressing views on the work that should be undertaken. Unfortunately, the views were often conflicting so it has not been possible to satisfy them all.

Although ALGOL 60 shows signs of being swamped by the expanding use of FORTRAN, and although ALGOL 68 exists, the remaining usage of the language is still significant and it remains much loved by its users.

The constancy of the language over many years should be regarded as one of its assets, not lightly to be disturbed. Changes should be kept to the minimum of necessary clarifications. Any large extensions, at this stage, would be doomed to be ignored, whereas we hope that the relatively small changes that we are suggesting may be incorporated into existing compilers.

It would seem wrong, after the Revised Report has existed unchanged for so many years, to try to force any changes by, for example, withdrawing IFIP recognition from the 1962 version in favour of any new proposals.

The suggestion, therefore, is that these proposals should be taken as defining a new language, to be called ALGOL 60.1, which, at least for awhile, would exist in parallel with Revised ALGOL 60, and reactions would be evaluated before reaching any final conclusion.

Two items that we have rejected, as being a little too radical, but that we should regard as strong candidates for consideration if it were decided to be bolder are (i) the iterative statement: while <Boolean expression> do <statement> (ii) the conditional string, defined by:

<simple string> ::= (<open string>) | (<string>)
<string> ::= <simple string>|<if clause><simple string>else<string>

We believe that there would be general (though not quite universal) rejoicing among ALGOL devotees if the extended input-output procedures of Knuth et al. (1964), and of ISO/R 1538 Part II B, were to be repudiated. In our commentary we have simply ignored them for the present.

We have not attempted to change the structure of the subsets, as defined in the ISO Recommendation, but in some instances (as detailed below) we believe that the present subset restrictions should apply to the full language (level 0). Also, having only six significant characters in an identifier at level 1 (ECMA subset with recursion) we feel is unduly restrictive. At levels 2 and 3 (the ECMA and IFIP subsets), it may be more difficult to ensure adherence to the additional restrictions than compile the full language.

This paper is in the form of a commentary on the Revised Report although most of these comments are expressed in the form of amendments. A booklet containing this paper, the Revised Report and our amendments applied to the Revised Report will be available[9].

A summary of our suggestions for language modification (as distinct from changes of wording without any change of intention) is as follows:

1. own variables are to be regarded as static. own arrays may only have fixed bounds. All own variables are to be initialised to zero or false.

2. The for statement is to be dynamic, but a step expression will be evaluated only once each time around the loop. The controlled variable cannot be a subscripted variable.

3. The controlled variable of a for statement will remain defined after exit from the loop.

4. Comments and strings are to consist of characters, not of ALGOL basic symbols, the characters allowed being implementation dependent.

5. Some new standard functions and procedures are introduced, including environmental enquiries and elementary transput.

6. Numerical labels are abandoned.

7. The effect of a go to statement leading to an undefined switch designator is to become undefined.

8. All formal parameters must be specified.

9. The exponentiation operator is to become undefined if both operands are of integer type, and the exponent is negative.

## Introduction

The Revised Report explicitly notes in the Introduction that five issues have been left unresolved and await further clarification. Our views on these matters are as follows:-

## Side effects of functions

Side effects of functions should be permitted without restriction, since it does not seem feasible to outlaw foolish uses without at the same time outlawing sensible uses. It is the programmer's responsibility not to employ the foolish uses.

It should be noted, in particular, that the Revised Report does not always specify the order in which expressions, or primaries within an expression, are to be evaluated. For instance, 3.3.5 specifies the order of execution of operations, but leaves undefined the order of evaluation of the primaries for those operations.

If different permitted orders of evaluation will produce different results, due to the action of side effects, then the action of the program must be regarded as undefined, in the sense of the footnote to the Revised Report, section 1. It should be noted that in the evaluation of a simple expression (either Boolean or arithmetic) all the primaries of the expression must be evaluated unless a jump out of a function is taken. A primary may contain expressions. The evaluation of a primary does not necessarily require the evaluation of all such expressions.

## The 'call by name' concept

There appears to be a need to modify to only a minor extent the detailed description of the execution of a procedure statement in 4.7. The exact effect of the call-by-name mechanism is there defined. See the commentary on 4.7.3.2 for the detailed amendment.

## Own: static or dynamic

The static interpretation of own is now accepted as standard. Ehat is to say: an own variable behaves exactly as if it had been declared in a block head immediately preceding the program, except that it is accessible only within its own scope. An extra end, corresponding to this fictitious block head, is assumed to follow the final end of the program. Possible conflicts between identifiers, resulting from this process, are resolved by suitable systematic changes of the identifiers involved.

It follows that: (i) an own variable, declared in a block within a procedure, which is called from different parts of the program, represents the same variable every time, not a separate variable for each place of call; (ii) an own variable, declared within a procedure that is activated recursively, represents the same variable at every level of the recursion; (iii) if a complete program is labelled, a go to leading to this label does not affect the values of own variables.

Furthermore, we recommend that this fictitious block should serve not only to declare any own variables, but also to assign initial values to them. All integer and real own variables should be assigned the value 0, while all Boolean own variables should be assigned the value false.

The bounds of an own array must be of the form <integer>. The second example of 5.2.2 must therefore be regarded as incorrect.

For statement: static or dynamic

The dynamic interpretation of the for statement has become accepted as standard, to such an extent that to many ALGOL 60 users it comes as a severe shock to be told that the Revised Report does not specify that this is the required interpretation. Having accepted the dynamic version, however, it still needs to be settled whether the step-expression has to be evaluated more than once per cycle, when a step-until element is being executed. The exact meaning of a subscripted controlled variable is also a matter of difficulty. It is now to be regarded as standard that the step expression should be evaluated once only per cycle, and that subscripted controlled variables should be forbidden. See the commentary on 4.6 below for the detailed amendments.

Conflict between specification and declaration

The Revised Report section 4.7.5 requires that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. This compatibility is defined by means of a table which appears under the commentary on that section.

In addition, the Introduction recognizes three different levels of language, Reference, Publication and Hardware. We propose that these should be reduced to Reference and Hardware only.

Publication language

The concept of publication language should no longer be recognised. It has become the universal practice that ALGOL 60 publications use reference language, with occasional minor variations in representation. These variations however (such as and for $\wedge$, or * for $\times$) are rarely, if ever, those recommended in the Revised Report for publication language.

Furthermore the wording of the Revised Report does not agree with what was presumably the intention, since removal of the upward arrow, as well as raising the exponent, was surely intended for exponentiation.

There is also an ambiguity introduced, since in reference language 2&5

is a number of real type, whereas 2*10↑5 is an expression of integer type. Yet both become $2*10^5$ in publication language.

## 1 Structure of the language

### The environmental block

A program is always considered to be contained within an additional level of block structure. This block is called the environmental block, and contains declarations of standard functions, input and output procedures, and possibly other procedures to be made available without declaration within the program as well as the fictitious declaration of own variables.

The environmental block includes declarations of at least the following procedures:

```
abs, iabs, sign, entier,
sqrt, sin, cos, arctan, ln, exp,
maxreal, minreal, maxint, epsilon,
fault, stop,
insymbol, outsymbol, inreal, outreal, ininteger,
        outterminator, outinteger, outstring, length.
```

It should be noted that since the environmental block is simply an ALGOL block, these identifiers may be redeclared within any other block if desired, with the usual scope rules applying.

The penultimate paragraph of section 1 should be amended to read:

'A program is a block or a compound statement that is contained only within a fictitious block, always assumed to be present, called the environmental block, and that makes no use of statements not contained within itself, except that it may invoke such procedure identifiers and function designators as may be assumed to be declared in the environmental block.

The environmental block contains procedure declarations of standard functions, input and output operations, and possibly other operations to be made available without declaration within the program. It also contains the fictitious declaration, and initialisation, of own variables (see section 5).'

The fictitious structure surrounding the program is:

```
begin
 <declaration of standard functions and procedures>;
 <fictitious declaration of own variables>;
 <initialisation of own variables>;
 <program>;
Ω :
end
```

where Ω is a label that is not accessible within the program but may be used by standard functions or procedures. Note that with this amendment the program 'sin: begin end' is no longer valid.

2 Basic symbols,identifiers,numbers and strings. Basic concepts

## 2.3 Delimiters

### Footnote concerning do

The footnote to 2.3, and the symbol that refers to this footnote (at the end of the definition of <sequential operator>), should both be deleted. It is unnecessary and confusing to readers who have no knowledge of the preliminary report, and also causes unnecessary ambiguity in the interpretation of the metalinguistic formulae. How can one tell that 'do "' (in the Comp.J. version), 'do $^{7}$' (in the Comm. ACM. version), 'do $^{2}$' (in the Num. Math. version), or 'do *' (in the ISO version) is not the required basic symbol?

### Space symbol

In line with the other modifications concerning strings (see 2.6), there is now no need for the space symbol in the Reference Language. Hence .| can now be deleted from the list of separators in 2.3. However, it is recommended that a visible character is used to represent a space so that typographical features are ignored throughout the language.

### Characters in comments

Section 2.3 allows only basic symbols within comments, although most compilers allow any hardware character and published ALGOL 60 often allows anything except semicolon. Indeed, the Revised Report examples contain several additional characters.

The relevant part of 2.3 should now read:

| 'The sequence | is equivalent to |
|---|---|
| ;comment <any sequence of zero or more characters not containing ;>; | ; |
| begin comment <any sequence of zero or more characters not containing ;>; | begin |
| end <any sequence of zero or more basic symbols not containing end or else or ;> | end |

This permits any characters after comment. It should be noted that the third type of comment (following end) is still restricted, since seeking for end or ; or else is more difficult for a compiler than merely seeking for ;.

### 2.6.1 Syntax

ALGOL 60 is not, and is not intended to be, a string manipulation language. The only use of strings is in communication to and from foreign media. It must be recognised that such foreign media deal in characters, not in ALGOL basic symbols. To be useful, the concept of a string must be put in touch with reality and be defined in terms of characters.

Characters are already recognised as existing in section 2.1 which says that the 'alphabet may ... be ... extended with any other distinctive character'. What characters are available must be a matter of hardware representation and be left undefined by the reference language just as 'code' is (see 5.4.6), except in insisting that string quotes must match, so that the end of a string can be detected.

To conform with the suggested change in strings to a sequence of characters and also to clarify the definition of <open string>, the syntax now becomes:-

<proper string> ::= <any sequence of characters not containing
( or ) >|<empty>
<open string> ::= <proper string>|<open string><string><proper string>

### 2.6.2 Examples

The character ., which is not now a basic symbol, is used to represent the position in a string at which a space is required.

### 2.6.3 Semantics

This section should now read:-

'In order to enable the language to handle sequences of characters the string quotes ( and ) are introduced.

The characters available within a string are a question of hardware representation, and further rules are not given in the reference language. However it is recommended that, in strings as elsewhere, typographical features such as blank space or change to a new line should have no significance, and that the character . should be used to represent a space.

Strings are used as actual parameters of procedures (see Sections 3.2 Function designators and 4.7 Procedure statements).'

## 3 Expressions

In the introduction to this section, the list of constituents of expressions omitted labels and switch designators. The second sentence should therefore read: 'Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, labels, switch designators, and elementary arithmetic, relational, logical, and sequential operators.'

3.1.3 Semantics

Add to this section:

'The value of a variable, not declared own, is undefined from entry into the block in which it is declared until an assignment is made to it.'

This brings variables into line with function values (see 5.4.4).

3.2.4 Standard functions

Replace the existing sections 3.2.4 and 3.2.5 by

'3.2.4 Standard functions and procedures

Certain standard functions and procedures are declared in the environmental block with the following procedure identifiers:
abs, iabs, sign, entier, sqrt, sin, cos, arctan, ln, exp, insymbol, outsymbol, length, outstring, outterminator, stop, fault, ininteger, outinteger, inreal, outreal, maxreal, minreal, maxint, and epsilon.

For details of these functions and procedures, see the specification of the environmental block given as Example 3, at the end of the report.'

The identifiers maxreal, minreal, maxint, and epsilon define functions, not standard variables, partly to avoid introducing a new concept unnecessarily, but mainly so as to make it impossible to assign to them.

3.2.5 Transfer functions

As with the other standard functions 'entier' must be provided in the environmental block and is not just a recommendation.

Section 3.2.5 should be deleted, since its purpose is now included in the new version of 3.2.4 given above.

3.3 Arithmetic expressions

3.3.3 Semantics

The largest arithmetic expression

The word 'longest' should be substituted for 'largest' in '(the largest arithmetic expression found in this position is understood)', since 'largest' might be taken as referring to the value of the expression.

The final sentence of this section should be deleted. It is incorrect since

else <simple arithmetic expression>

must not be followed by a further else, whereas

else if true then <simple arithmetic expression>

must be followed by a further else. The two constructions are therefore not equivalent.

It should be replaced by
'If none of the Boolean expressions has the value true, then the value of the arithmetic expression is the value of the expression following the final else'.

### 3.3.4.2 Division operators

Amend the first sentence by changing 'denote division, to be understood' to read 'denote division. The operations are undefined if the factor has the value zero, but are otherwise to be understood'.

It should be noted that the word 'mathematically', in the definition of integer division, is intended to signify that the specified operations are to be performed without rounding error.

The result of integer division can be given by means of a function. Hence the words 'mathematically defined as follows:' to the end of the section should be replaced by 'if a and b are of integer type, then the value of a div b is given by the function:

```
integer procedure div(a, b); value a, b;
     integer a, b;
     if b = 0 then
          fault( (div.by.zero) , a)
     else
          begin integer q, r;
          q := 0;  r := iabs(a);
          for r := r - iabs(b) while r ≥ 0 do
               q := q + 1;
          div := if a < 0 equiv b > 0 then -q else q
          end div'
```

It should be noted that although real expressions could be used as arguments to the procedure div, the operator div is permitted only with operands of type integer. It also should be noted that div is not a standard function.

### 3.3.4.3 Exponentiation operator

Rather than give a table of values given by this operator, it seems more appropriate to define the values by means of algorithms. To achieve this, the second half of this section starting 'Writing i for a number ...' can be replaced by :-

'If r is of real type and x of either real or integer type, then the value of x↑r is given by the function:

```
real procedure expr(x, r); value x, r;
      real x, r;
      if x > 0.0 then
           expr := exp(r*ln(x))
      else if x = 0.0 and r > 0.0 then
           expr := 0.0
      else
           fault( (expr.undefined) , x)
```

If n is of integer type and x of real type, then the value of x↑n is given by the function:

```
real procedure expn(x, n); value x, n;
      real x;  integer n;
      if n = 0 and x = 0.0 then
           fault( (0.0↑0) , x)
      else
           begin
           real result;  integer i;
           result := 1.0;
           for i := iabs(n) step -1 until 1 do
                 result := result*x;
           expn := if n<0 then 1.0/result else result
           end expn
```

If i and j are both of integer type, then the value of i↑j is given by the function:

```
integer procedure expi(i, j);  value i, j;
       integer i, j;
       if j < 0 or i = 0 and j = 0 then
            fault( (expi.undefined), j)
       else
            begin
            integer k, result;
            result := 1;
            for k := 1 step 1 until j do
                 result := result * i;
            expi := result
            end expi
```

The call of the procedure fault denotes that the action of the program is undefined. The numerical accuracy of particular implementations of this operator should be no worse than that produced by the above algorithms.'

The Revised Report contains a difficulty with this operator in that the type of <integer>↑<integer> depends upon the sign of the exponent. The above implementation is undefined if the factor and primary are of type integer and the primary is negative. If it is desired that a real result should be produced then i↑j can be written as float(i)↑j where float is a function which gives the real value as in the assignment float := i. It should be noted that float is not a standard function.

In many ways a much neater solution would be to have two different symbols, for real exponentiation and integer exponentiation, in a similar manner to real and integer division, but the above seems the best compromise, as we do not consider that it would be wise to introduce any new basic symbol.

3.3.4.4 Type of a conditional expression

Since the type of a conditional expression is not specified in the Revised Report, a new section is required thus:-

The type of an arithmetic expression of the form
if B then SAE else AE
does not depend upon the value of B. The expression is of type real if either SAE or AE is real and is of type integer otherwise.

3.3.5 Precedence of operators

It should be noted that although the precedence of operators determines the order in which the operations are performed, the order of evaluation of the primaries for these operations is not defined.

3.3.6 Arithmetics of real quantities

The reference to 'hardware representations' should be replaced by 'implementations', since elsewhere in the Revised Report 'hardware representation' refers to the representation of basic symbols.

3.4 Boolean expressions

3.4.5 The operators

Insert as the first sentence 'The relational operators <, ≤, =, ≥, > and ne have their conventional meaning (less than, less than or equal to, equal to, greater than or equal to, greater than, not equal to).'

3.5 Designational expressions

3.5.1 Syntax

Numerical labels

Numerical labels add in no way to the power or usefulness of the language although providing difficulties for the compiler-writer. They must now be regarded as obsolete in the full language as well as in the subsets. The syntax should now be

<label> ::= <identifier>

3.5.2 Examples

To conform to the change in labels, in the first and last examples, replace 17 by L17.

3.5.5 Unsigned integers as labels

Delete this section.

4 Statements

4.1 Compound statements and blocks

4.1.3 Semantics

Replace the last sentence of the second paragraph by:

'A label is said to be implicitly declared in this block head, as distinct from the explicit declaration of all other local identifiers. In this context a procedure body, or the statement following a for clause, must be considered as if it were enclosed by begin and end and treated as a block. A label that is not within any block of the program (nor within a procedure body, or the statement following a for clause) is implicitly declared in the head of the environmental block.'

4.2 Assignment statements

4.2.3 Semantics

Amend 'the body of a procedure defining the value of a function designator' to read 'the body of the procedure defining the value of the function designator denoted by that identifier.' This ensures that an assignment to a function can occur only within that function.

To conform to the requirement on access to a subscripted variable add to this section:

'If assignment is made to a subscripted variable, the values of all the subscripts must lie within the appropriate subscript bounds. Otherwise the action of the program becomes undefined.'

4.2.4 Types

Replace the wording 'equivalent to entier (E + 0.5)' by 'which is the largest integral quantity not exceeding E + 0.5 in the mathematical sense (i.e. without rounding error).'

4.3 Go to statements

4.3.2 Examples

The labels 8 and 17 be must replaced by L8 and L17 respectively since integer labels are no longer permitted.

4.3.5 Go to an undefined switch designator

Replace this section by:

'A go to statement is undefined if the designational expression is a switch designator whose value is undefined.'

4.4 Dummy statements

4.4.2 Examples

Amend the second example to read

```
begin statements; John: end
```

This is necessary since '...' is not valid ALGOL 60.

4.5 Conditional statements

4.5.3.1 If statement

Reword this section as follows:

'An if statement is of the form

```
if B then Su
```

where B is a Boolean expression and Su is an unconditional statement. In execution, B is evaluated; if the result is true, Su is executed; if the result is false, Su is not executed.

If Su contains a label, and a go to statement leads to the label, then B is not evaluated, and the computation continues with execution of the labelled statement.'

4.5.3.2 Conditional statement

Reword this section as follows:

'Three forms of unlabelled conditional statement exist, namely:

```
if B then Su
if B then Sfor
if B then Su else S
```

where Su is an unconditional statement, Sfor is a for statement and S is a statement.

The meaning of the first form is given in 4.5.3.1.

The second form is equivalent to

```
if B then begin Sfor end
```

The third form is equivalent to

```
    begin
    if B then begin Su; goto L4 end;
    S;
L4: end
```

If the use of L4 causes any clash of identifiers it must be systematically changed to some other identifier - in particular, if S is conditional, and also of this form, a different label must be used in following the same rule.'

4.5.4 Go to into a conditional statement

Delete the last three words and substitute 'execution of a conditional statement.'

4.6 For statements

The exact interpretation of the ALGOL 60 for loop mechanism is controversial. The method given below has the advantage of being expressed in ALGOL 60.

4.6.1 Syntax

Replace the syntax of <for clause> by

<for clause> ::= for <variable identifier> := <for list> do

4.6.3 Semantics

Replace this section by:

'A for clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable (the variable after for). The controlled variable must be of real or integer type.'

4.6.4 The for list elements

Replace this section by:

'If the for list contains more than one element then

for V := X, Y do S where X is a for list element, and Y is a for list (which may consist of one element or more), is equivalent to

```
begin
procedure S1;  S;
for V := X do S1;
for V := Y do S1
end
```

Repeated use of this rule enables any for statement with n elements to be changed to n for statements with one element each. If the use of S1 causes any clash of identifiers it must be systematically changed to some other identifier.'

4.6.4.1 Arithmetic expression element

Replace this section by:

'If X is an arithmetic expression

```
for V := X do S
```

is equivalent to

```
begin
V := X;  S
end
```

where S is treated as if it were a block (see 4.1.3).'

4.6.4.2 Step-until element

Replace this section by:

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

is equivalent to

```
begin <type of B> D;
V := A;  D := B;
L1: if (V-C)*sign(D) ≤ 0 then
        begin
        S;  V := V+D;
        D := B;  goto L1
        end
end
```

where S is treated as if it were a block (see 4.1.3).

In the above, <type of B> must be replaced by real or integer according to the type of B. If the use of D, or of L1, causes any clash of identifiers, it must be systematically changed to some other identifier.'

If it were decided to allow subscripted controlled variables, the method should be:

```
for V[i] := A step B until C do S
```

is to mean

```
begin <type of B> D; integer j;
j := i;  V[j] := A;  D := B;
L1: if (V[j] - C) * sign(D) ≤ 0 then
        begin
        S; j := i;
        V[j] := V[j] + D;  D := B;
        goto L1
        end
end
```

and similarly with controlled variables having more than one subscript.

4.6.4.3 While element

Replace this section by:

'for V := E while F do S

is equivalent to

```
        begin
L3:     V := E;
        if F then
                begin
                S;  goto L3
                end
        end
```

where S is treated as if it were a block (see 4.1.3). If the use of L3 causes any clash of identifiers it must be systematically changed to some other identifier.'

4.6.5 The value of the controlled variable upon exit

Replace this section by:

'Upon exit from the for statement, either through a go to statement, or by exhaustion of the for list, the controlled variable retains the last value assigned to it.'

4.6.6 Go to leading into a for statement

Replace this section by:

'The statement following a for clause always acts like a block, whether it has the form of one or not. Consequently the scope of any label within this statement can never extend beyond the statement.'

In general the rules given above are merely a tidying operation, removing certain ambiguities and uncertainties. However, there are some minor changes in what is to be regarded as correct ALGOL 60, as follows:

(i) for v[i] := <for list> do becomes incorrect, since a subscripted controlled variable is not allowed;

(ii) for j := A[i] while j=0 do i := i+1; examine(j) becomes correct, since j is defined after the for statement;

(iii) for j := k, m, n do q[j] := j; i := j becomes correct. j has the value n after the for statement;

(iv)

```
begin switch m := a,b;
.................
.................
for ....... do
        begin
        ..............
        ..............
        goto m[i];
        ..............
 a:     ..............
 b:     ..............
        end
end
```

becomes incorrect, since the scope of a and b does not extend to the switch declaration. The switch should be declared after the second begin instead of after the

first;

(v)

```
          .............
          .............
          for ........ do
                 begin
                 .............
                 .............
            m:   .............
                 .............
                 end;
          .............
     m:   .............
          .............
```

becomes correct, since the scope of the inner m does not extend beyond the for statement;

(vi) If the controlled variable is a name parameter, then the rules for a procedure call (see 4.7.3.2) prohibit the actual parameter from being a subscripted variable. The check for this restriction need be performed only on initial entry to the loop and not every time round the loop;

4.7 Procedure statements

4.7.3.2 Name replacement (call by name)

In the first sentence replace 'wherever syntactically possible' by 'if it is an expression but not a variable'. This avoids the difficulty with the existing wording that if procedure A has a parameter, that is passed to procedure B, procedure B may be unable to assign to it,since it may have been syntactically possible within A to put parentheses around it.

4.7.5 Restrictions

Amend the second sentence of the second paragraph to read: 'Some important particular cases of this general rule, and some additional restrictions, are the following:'

4.7.5.4

Add to this section:
'A label may be called by value, even though variables of type label do not exist.'

This facility is necessary at level 3, to allow a switch designator to be used as the actual parameter.

Add to this section:
'The correspondence between actual and formal parameters should be in accordance with the following table:

| FORMAL PARAMETER | MODE | VALID ACTUAL PARAMETERS LEVEL 0 | LEVELS 1,2 | LEVEL 3 |
|---|---|---|---|---|
| integer | value | ae | ae | ae |
| | name | ae* | ie* | is |
| real | value | ae | ae | ae |
| | name | ae* | re* | rs |
| Boolean | value | be | be | be |
| | name | be* | be* | bs |
| label | value | de | de | l,sd |
| | name | de | de | l |
| integer array+ | value | aa | ia | ia |
| | name | ia | ia | ia |
| real array+ | value | aa | ra | ra |
| | name | ra | ra | ra |
| Boolean array+ | value | ba | ba | ba |
| | name | ba | ba | ba |
| typeless procedure+ | name | ap,bp,tp | tp | tp |
| integer procedure+ | name | ap | ip | ip |
| real procedure+ | name | ap | rp | rp |
| Boolean procedure+ | name | bp | bp | bp |
| switch | name | sw | sw | sw |
| string | name | st | st | st |

key:designational:d
arithmetic: a
integer: i
real: r
Boolean: b
typeless: t
expression: e
simple variable: s
array: a
procedure: p

label: l
switch designator: sd
switch: sw
actual string or string identifier: st

* Where an assignment is made to the formal parameter, either explicitly in the body of the procedure, or implicitly by means of a further procedure call in which such an assignment is made, the actual parameter must be a variable.
+ With an array parameter, the number of subscripts appearing in any of

its subscript lists must agree with those of the actual parameter. Similarly, the number, kind and type of the parameters of a formal procedure parameter must agree with the actual parameter.

In a procedure call, for each corresponding pair of actual and formal parameters, the actual parameter A must satisfy the rules in the above table, depending on the type and mode of the formal parameter F.

If A is itself a formal parameter, it must satisfy the rules above depending solely on its specification, irrespective of the nature of its own actual parameter. Thus, if type conversion (e.g. integer-to-real ) is required by the parameter substitution, this process takes place independent of the type of the actual parameter substituted for the formal parameter which is itself the actual parameter in the parameter substitution under consideration.'

The following example should make this clear:

```
begin
      real x, y;
      procedure p(i); integer i;
            q(i);
      procedure q(z); real z;
            y := z;
      x := 6.2;
      p(x)
end
```

The statement 'y := z' requires the evaluation of the actual parameter 'i' in p. This in turn requires the evaluation of the actual parameter 'x' in the outer block. A type conversion (real to integer) is invoked, giving 'i' a value of 6, and a further conversion (integer to real), giving 'z' the value 6.0. Hence, y is assigned the value 6.0.

### 4.7.9 Standard procedures

The Revised Report did not contain any procedures to handle input-output. To rectify this, and to facilitate the handling of error conditions, ten standard procedures are defined below. With the exception of outterminator, fault and stop, all these procedures appear in the IFIP recommendations for input-output[5]. However the IFIP procedures inarray and outarray have not been implemented, since their effect can be achieved by means of the procedures inreal and outreal within suitable for statements. The new section, defining these procedures is:-

'Ten standard procedures are defined, which are declared in the environmental block in an identical manner to the standard functions. These procedures are:- insymbol, outsymbol, outstring, ininteger, inreal, outinteger, outreal, outterminator, fault and stop. The input-output procedures identify physical devices or files by means of channel numbers which appear as the first parameter. The method by which this identification is achieved is outside the scope of this report. Each channel is regarded as containing a sequence of characters, the basic method of accessing or assigning these characters being via the procedures insymbol and outsymbol.

The procedures inreal and outreal are converses of each other in the sense that a channel containing characters from successive calls of outreal can be re-input by the same number of calls of inreal, but some

accuracy may be lost. The procedures ininteger and outinteger are also a pair, but no accuracy can be lost. The procedure outterminator is called at the end of each of the procedures outreal, outinteger and outstring. Its action is machine dependent but it must ensure separation between successive output of numeric data.

These additional procedures are given as examples to illustrate the environmental block at the end of this report.'

## 5 Declarations

Delete the last two sentences ('Apart from labels ... one block head') and substitute the following:

'Apart from labels, formal parameters of procedure declarations, and identifiers declared in the environmental block, each identifier appearing in a program must be explicitly declared within the program.

No identifier may be declared either explicitly or implicitly (see 4.1.3) more than once in any one block head.'

## 5.1 Type declarations and 5.2 Array declarations

The syntax of 5.2.1 allows array, to be understood (5.2.3.3) as meaning real array. Yet own real array must be written in full, the abbreviation own array being prohibited.

To allow own array the following amendments should be made.

In 5.1.1 delete the definition of <local or own type> and <type declaration> and substitute:

<type declaration> ::= <type><type list>|own<type><type list>

In 5.2.1 delete the definition of <array declaration> and substitute:

<array declarer> ::= array<array list>|<type>array<array list>
<array declaration> ::= <array declarer>|own<array declarer>

### 5.1.3 Semantics

Because of the restrictions imposed upon exponentiation at level 3, a real variable cannot always be replaced by an integer variable. There are also difficulties at all levels with procedure parameters and hence, at all levels, the second paragraph of this section should be omitted.

### 5.2.2 Examples

The second example should be deleted, as an own array may only have constant bounds.

5.2.4 Lower upper bound expressions

Problems arise through the scope of identifiers appearing in these expressions which we hope are clarified by the following changes.

Replace section 5.2.4.2 by:

'5.2.4.2 The expression cannot include any identifier that is declared, either explicitly or implicitly (see 4.1.3), in the same block head as the array in question. The bounds of an array declared as own may only be of the syntactic form integer (see 2.5.1).'

Section 5.2.4.3 specifies the conditions under which an array is defined. An undefined array, in the sense of this section, should not be regarded as a fault but merely as giving an array of zero elements. To ensure this interpretation, add to this section 'If any lower subscript bound is greater than the corresponding upper bound, the array has no elements.'

The array identifier may then be used (for example as an actual parameter, even if called by value), but any reference to an element of the array will be incorrect.

Thus:

```
begin array A[1:n]; integer i;
.........
.........
for i := 1 step 1 until n do
          operate( A[i] );
.........
.........
end
```

is valid even if n=0. The array will not exist, but neither will its elements be accessed.

5.2.5 The identity of subscripted variables

This section should be deleted. The second sentence is no longer relevant, whereas the meaning, if any, of the first sentence is unclear.

5.4.3 Semantics

Add to the end of this section:

'No identifier may appear more than once in any one formal parameter list, nor may a formal parameter list contain the procedure identifier of the same procedure heading.'

5.4.4 Values of function designators

Modify 'in a left part' (in each of two places) to read 'as a left part'. This is necessary as a function designator can appear in a subscript expression in a left part.

A difficulty arises with a go to leading out of a function designator since if this jump is executed, no value for the function is defined. To

clarify that such jumps are permitted, at the end of the section add the following words:
'If a go to statement within the procedure, or within any other procedure activated by it, leads to an exit from the procedure, other than through its end, then the execution, of all statements that have been started but not yet completed and which do not contain the label to which the go to statement leads, is abandoned. The values of all variables that still have significance remain as they were immediately before execution of the go to statement.

If a function designator is used as a procedure statement, then the resulting value is lost, but such a statement may be used, if desired, for the purpose of invoking side effects.'

Some examples of jumping out of a function are:

```
(i)      j := 3;
         j := p(L);
         ........
     L:  ........
```

If the jump is taken, j will still have the value 3 when L is reached.

```
(ii)   procedure q(k);
         value k; integer k;
         begin
         .........
         .........
         end q;
         .........
         .........
         q(p(L));
         .........
     L:  .........
```

If the jump is taken, none of the statements of q will be performed.

```
(iii)   i := m[k] := n[p(L)] := s[t] := j := 3;
              ........
         L:   ........
```

If the jump is taken, none of the variables will have the value 3 assigned to it. Any side effects due to evaluation of k will have been performed; any side effects due to evaluation of t will not (see 4.2.3.1, 4.2.3.2 and 4.2.3.3).

```
(iv)       L: ........
              ........
           M: begin array a[ 1:p(L) ];
              ........
              ........
              end
```

If the jump is taken, execution of the block labelled M is abandoned. Note that, by 5.2.4.2, L can only be outside the block (thank goodness).

### 5.4.5 Specifications

Incomplete specification of parameters appears to be inconsistent with the spirit of ALGOL 60, since with declarations, explicit type indications are required. Moreover, incomplete specification causes significant

definition and implementation problems. The table given under 4.7.5.5 would no longer specify adequately the valid correspondence between formal and actual parameters. Hence we believe section 5.4.5 should be replaced by: 'In the heading a specification part, giving information about the kinds and types of the formal parameters must be included. In this part no formal parameter may occur more than once.'

5.4.6 Code as procedure body

In the final sentence change 'hardware representation' to 'implementation'.

Examples

As a further example of the use of ALGOL 60, the structure of the environmental block is given in detail.

EXAMPLE 3

```
begin

    comment   Simple functions;

    real procedure abs(E);
        value E;
        real E;
        abs :=
            if E ≥ 0.0 then
                E
            else
                - E;

    integer procedure iabs(E);
        value E;
        integer E;
        iabs :=
            if E ≥ 0 then
                E
            else
                - E;

    integer procedure sign(E);
        value E;
        real E;
        sign :=
            if E > 0.0 then
                1
            else if E < 0.0 then
                - 1
            else
                0;

    integer procedure entier(E);
        value E;
        real E;
```

```
    comment   entier := largest integer not greater
        than E, i.e. E - 1 < entier ≤ E;

    begin
    integer j;
    j := E;
    entier :=
        if j > E then
            j - 1
        else
            j
    end entier;


comment   Mathematical functions;

real procedure sqrt(E);
    value E;
    real E;
    if E < 0.0 then
        fault( (negative.sqrt) , E)
    else
        sqrt := E↑0.5;

real procedure sin(E);
    value E;
    real E;

    comment   sin := sine of E radians;
    <body>;

real procedure cos(E);
    value E;
    real E;

    comment   cos := cosine of E radians;
    <body>;

real procedure arctan(E);
    value E;
    real E;

    comment   arctan := principal value, in radians,
        of arctangent of E, i.e. -pi/2 ≤ arctan ≤ pi/2;
    <body>;

real procedure ln(E);
    value E;
    real E;

    comment   ln := natural logarithm of E;

    if E ≤ 0.0 then
        fault( (ln.not.positive) , E)
    else
        <statement>;

real procedure exp(E);
    value E;
```

```
    real E;

    comment   exp := exponential function of E;

    if E > ln(maxreal) then
        fault( (overflow.on.exp) , E)
    else
        <statement>;


comment   Input - output procedures;

procedure insymbol(channel, str, int);
    value channel;
    integer channel, int;
    string str;

    comment   Set int to value corresponding to the first
        position in str of current character on channel. Set
        int to zero if character not in str, unless it is
        a non-printing character, in which case set int to a
        negative integer associated with the character. Move
        channel pointer to next character;

    <body>;


procedure outsymbol(channel, str, int);
    value channel, int;
    integer channel, int;
    string str;

    comment   Pass to channel the character in str,
        corresponding to the value of int. If int is
        negative, pass the associated non-printing character,
        where the association is the same as for insymbol;

    if int = 0 or int > length(str) then
        fault( (character.not.in.string) , int)
    else
        <statement>;

integer procedure length(str);
    string str;

    comment   length := number of characters in the open
        string enclosed by the outermost string quotes;
    <body>;

procedure outstring(channel, str);
    value channel;
    integer channel;
    string str;
    begin
    integer m, n;
    n := length(str);
    for m := 1 step 1 until n do
        outsymbol(channel, str, m);
    outterminator(channel)
```

```
    end outstring;

procedure outterminator(channel);
    value channel;
    integer channel;

    comment   outputs a terminator for use after every
        string or number. To be converted into format
        control instructions in a machine dependent
        fashion. The terminator should be a space or a
        semicolon if ininteger and inreal are to be able
        to read representations resulting from outinteger
        and outreal;

   <body>;

procedure stop;

    comment  Ω is assumed to be the label of a dummy
        statement immediately preceding the end
        of the environmental block;

    goto Ω;

procedure fault(str, r);
    value r;
    string str;
    real r;

    comment   sigma is assumed to be an integer
        constant that denotes a standard output channel.
        The following calls of fault appear:
           integer divide by zero,
           undefined operation in expr,
           0.0 ↑ 0 in expn,
           undefined operation in expi,
        and in the environmental block:
           sqrt of negative argument,
           ln of negative or zero argument,
           overflow on exp function,
           illegal parameter for outsymbol,
           invalid character in ininteger(twice),
           invalid character in inreal(three times);

    begin
    outstring(sigma, (FAULT) );
    outstring(sigma, str);
    outreal(sigma, r);

    comment   Additional diagnostics may be output here;

    stop
    end fault;

procedure ininteger(channel, int);
    value channel;
    integer channel, int;

    comment int takes the value of an integer, as defined
```

```
        in 2.5.1, read from channel. Any number of spaces
        or other non-printing characters may precede
        the first visible character. The terminator of
        the integer may be either a space or other
        non-printing character or a semicolon (if other
        terminators are to be allowed, they may be added to the
        end of the string parameter of the call of insymbol.
        No other change is necessary);

    begin
    integer k, m;
    Boolean b, d;
    integer procedure ins;
        begin
        integer n;
        insymbol(channel, (0123456789-+.;), n);
        ins :=  if n < 0 then 13 else n
        end ins;

    for k := ins while k = 13 do
        ;
    if k < 1 or k > 13 then
        fault( (invalid.character) , k);
    if k > 10 then
        begin
        d := false;
        b := k = 12;
        m := 0
        end
    else
        begin
        d := b := true;
        m := k - 1
        end;
    for k := ins while k > 0 and k < 11 do
        begin
        m := 10 * m + k - 1;
        d := true
        end k loop;
    if d impl k < 13 then
        fault( (invalid.character) , k);
    int :=
        if b then
            m
        else
            - m
    end ininteger;

procedure outinteger(channel, int);
    value channel, int;
    integer channel, int;

    comment Passes to channel the characters representing
        the value of int, followed by a terminator;

    begin
    procedure digit(int);
        value int;
        integer int;
```

```
        begin
        integer j;
        j := int div 10;
        int := int - 10 * j;
        if j ne 0 then
            digit(j);
        outsymbol(channel, (0123456789), int + 1)
        end;

    if int < 0 then
        begin
        outsymbol(channel, (-), 1);
        int := - int
        end;
    digit(int);
    outterminator(channel)
    end outinteger;

procedure inreal(channel, re);
    value channel;
    integer channel;
    real re;

    comment re takes the value of a number, as
        defined in 2.5.1, read from channel. Except for
        the different definitions of a number and an
        integer the rules are exactly as for ininteger.
        Spaces or other non-printing characters may
        follow the symbol &;

    begin
    integer j, k, m;
    real r, s;
    Boolean b, d;
    integer procedure ins;
        begin
        integer n;
        insymbol(channel, (0123456789-+.&;), n);
        ins := if n < 0 then 15 else n
        end ins;

    for k := ins while k = 15 do
        ;
    if k < 1 or k > 15 then
        fault( (invalid.character) , k);
    b := k ne 11;
    d := true;
    m := 1;
    j :=
        if k < 11 then
            2
        else
            iabs(k + k - 23);
    r :=
        if k < 11 then
            k - 1
        else
            0.0;
    if k ne 14 then
```

```
        begin
        for k := ins while k < 14 do
            begin
            if k < 1 or k = 11 or k = 12
                or k = 13 and j > 2 then
                fault( (invalid.character) , k);
            if d then
                begin
                if k = 13 then
                    j := 3
                else
                    begin
                    if j < 3 then
                        r := 10.0 * r + k - 1
                    else
                        begin
                        s := 10.0↑( - m);
                        m := m + 1;
                        r := r + s * (k - 1);
                        d := r ne r + s
                        end;
                    if j = 1 or j = 3 then
                        j := j + 1
                    end
                end
            end k loop;
        if j = 1 and k ne 14 or j = 3 then
            fault( (invalid.character) , k)
        end;
    if k = 14 then
        begin
        ininteger(channel, m);
        r := (if j = 1 or j = 5 then 1.0 else r)
             * 10.0 ↑m
        end;
    re :=
        if b then
            r
        else
            - r
    end inreal;

procedure outreal(channel, re);
    value channel, re;
    integer channel;
    real re;

    comment Passes to channel the characters representing
        the value of re, followed by a terminator;

    begin
    integer n;
    n := entier(1.0 - ln(epsilon) / ln(10.0));
    if re < 0.0 then
        begin
        outsymbol(channel, (-), 1);
        re := - re
        end;
    if re < minreal then
```

```
    begin
    outstring(channel, (0.0) );
    end
else
    begin
    integer j, k, m, p;
    Boolean float, nines;
    m := 0;
    nines := false;
    for m := m + 1 while re ≥ 10.0 do
        re := re / 10.0;
    for m := m - 1 while re < 1.0 do
        re := 10.0 * re;
    if re ≥ 10.0 then
        begin
        re := 1.0;
        m := m + 1
        end;
    if m ≥ n or m < - 2 then
        begin
        float := true;
        p := 1
        end
    else
        begin
        float := false;
        p :=
            if m = n - 1 or m < 0 then
                0
            else
                m + 1;
        if m < 0 then
            begin
            outsymbol(channel, (0), 1);
            outsymbol(channel, (.), 1);
            if m = - 2 then
                outsymbol(channel, (0), 1)
            end
        end;
    for j := 1 step 1 until n do
        begin
        if nines then
            k := 9
        else
            begin
            k := entier(re);
            if k > 9 then
                begin
                k := 9;
                nines := true
                end
            else
                re := 10.0 * (re - k)
            end;
        outsymbol(channel, (0123456789), k + 1);
        if j = p then
            outsymbol(channel, (.), 1)
        end j loop;
    if float then
```

```
            begin
            outsymbol(channel, (&), 1);
            outinteger(channel, m)
            end
        else
            outterminator(channel)
        end
    end outreal;


comment   Environmental enquiries;

real procedure maxreal;
    maxreal := <number>;

real procedure minreal;
    minreal := <number>;

integer procedure maxint;
    maxint := <integer>;


comment   maxreal, minreal, and maxint are, respectively
    the maximum allowable positive real number, the
    minimum allowable positive real number, and the
    maximum allowable positive integer, such that any
    valid expression of the form
        <primary><arithmetic operator><primary>
    will be correctly evaluated, provided that each of the
    primaries concerned, and the mathematically correct
    result lies within the open interval (-maxreal,-minreal)
    or (minreal,maxreal) or is zero if of real type, or within
    the open interval (-maxint,maxint) if of integer
    type.
    If the result is of real type, the words 'correctly
    evaluated' must be understood in the sense of
    numerical analysis (see Revised Report 3.3.6);

real procedure epsilon;

    comment   The smallest positive real number such that the
        computational result of 1.0+epsilon is greater than 1.0
        and the computational result of 1.0-epsilon is less than
        1.0;

    epsilon := <number>;

comment In any particular implementation, further
    standard functions and procedures may be added here,
    but no additional ones may be regarded as part of the
    reference language;

<fictitious declaration of own variables>;
<initialisation of own variables>;

<program>;

Ω:
end
```

Notes on the standard procedures and functions

The above coding is only to be taken as definitive in terms of its effect on correct programs, ignoring those questions which are the domain of numerical analysis. For instance, a call of the procedure 'fault' indicates that the program is in error, and hence after detection of the error, different action may be taken than that indicated by the above coding. Actual implementations may produce better diagnostics than are possible to express conveniently in ALGOL 60.

The procedures sin, cos, arctan, ln, and exp have some coding omitted because their definition is clear and this report is not concerned with the methods used in the evaluation of these functions. The bodies of the procedures insymbol, outsymbol, length, outterminator, maxreal, minreal, maxint and epsilon are omitted because of their obvious machine dependence. The procedures insymbol and outsymbol are used on the assumption that the relevant 'ALGOL basic symbols' are single characters. Appropriate changes must be made if this is not the case, although the only likely exception is the use of & in 'inreal' and 'outreal'.

Naturally, implementations should gain significantly in performance over the coding given above. In particular, the simple functions may be performed by open code, the variable n in outreal can be assigned the appropriate constant value, the procedure identifiers maxreal etc can be replaced by a constant value and the recursive nature of the procedure digit can be avoided. Also, the numeric properties of the procedures inreal and outreal can be enhanced by the use of double length working, although these procedures have been tested and found to be adequate (within the constraints of single precision).

Index

The following corrections should be made to the index of the Revised Report:-

. delete entry to conform with amendments.

<arithmetic expression> delete 'synt 3.3.1' as this appears under def.

<array declarer> add entry containing 'def 5.2.1'

<local or own type> delete entry.

<procedure identifier> insert 4.2.1 under synt.

<simple arithmetic expression> insert 'synt 3.4.1'.

space delete 'def 2.3'

<type> add 'synt 5.2.1'

<unsigned integer> delete '3.5.1.'

<variable> delete '4.6.1,'

<variable identifier> insert 'synt 4.6.1'

The documents used to construct this commentary are too numerous to list, but the principle references are:

[1] Naur, P (Editor) Revised Report on the Algorithmic Language ALGOL 60,
Comm ACM, Vol 6 (1963), p1
Comp J, Vol 5 (1963), p349
Num Math, Vol 4 (1963), p420

[2] Report on Subset ALGOL 60 (IFIP),
Num Math, Vol 6 (1964), p454
Comm ACM, Vol 7 (1964), p626

[3] ECMA Subset of ALGOL 60,
Comm ACM, Vol 6 (1963), p595
European Computer Manufacturers Association (1965) ECMA Standard for a Subset of ALGOL 60.

[4] ISO/R 1538, Programming Language ALGOL (1972)

[5] Report on Input-Output Procedures for ALGOL 60 (IFIP),
Num Math, Vol 6 (1964), p459
Comm ACM, Vol 7 (1964), p628

[6] Knuth, D.E. et al, A Proposal for Input-Output Conventions in ALGOL 60,
Comm ACM, Vol 7 (1964), p273

[7] Knuth, D.E. The Remaining Trouble Spots in ALGOL 60
Comm ACM, Vol 10 (1967), p611

[8] Suggestions on the ALGOL 60 (Rome) Issues,
Comm ACM, Vol 6 (1963), p20

[9] A booklet on ALGOL 60,
Joint IFIP/NPL Publication, to be prepared.

AB38.4.1 A View on Simulation in Algol 68

D C S Shearn
University of Sheffield

A number of articles have appeared recently whose motivation has been in part the desire to carry out discrete event simulation in Algol 68 (Levinson, AB 36.4.2; Lindsey, AB 37.4.2 and AB 37.4.3). The purpose of this note is to comment on these articles, to mention some of the features of a simulation package that has been implemented in Algol 68, and to suggest a direction of development that would be useful to simulators. The barber shop example of earlier articles is used.

## Requirements of a Simulation Package

The basic requirements of a simulation package in any language are easily stated. In addition to the facilities normally found in a good high level language, there should be easily used features for

1. the description of entities (barbers, customers, etc.)
2. an executive to control the passage of simulated time and the execution of events
3. list processing/queueing/set handling
4. random sampling from various distributions
5. data collection within the simulation, its analysis and presentation
6. monitoring the progress of the simulation

Items 4 and 5 are easily catered for in Algol 68. For item 6, it must be up to the simulator to include specific monitoring statements within his program, and to make the best use he can of the facilities provided by the implementation.

It is items 1,2 and 3 and the inter-relationships between them that are the main distinguishing features of the requirements of a simulation package.

## The Simulation Executive

There are basically four approaches to designing an executive for a simulation package. These are

1. an event scheduling system (e.g. Simscript)
2. an event scanning system (e.g CSL)
3. a hybrid approach having features of 1 and 2 and sometimes referred to as the three phase system
4. a process control system (e.g. Simula)

The first three can be implemented reasonably satisfactorily in Algol 68, and the author has done this within a single package (1). However, from the simulator's point of view, these approaches lack some of the elegance and power inherent in the process control approach whereby related events can be joined together to form a single process or activity. Levinson's ingenious approach to the process control method using parallel processing must therefore be viewed with considerable interest. However, even if there were to be compilers

available which had implemented parallel processing, there would still be serious practical problems in using his procedures. These are

1. The parallel nature of the events within processes can produce different results for the same program even when all the data and random numbers are the same. For instance, if a barber finishes smoking at the same instant of simulated time that a customer enters the shop, there being no other waiting customers, on some occasions the barber would start another smoke, and on others he would serve the customer. This type of randomness, which is outside the simulator's control, can make debugging particularly difficult.

2. Difficulties arise when the barbers do not smoke, but merely sit around waiting for customers. One can get round this by making each idle barber have a "pseudo" smoke lasting just one time unit, after which they must all go through the ritual of seeing if there is a customer for them to serve. There are other ways of trying to get round this difficulty, but they all seem to lead to the same type of inefficiency.

3. In some simulations, it may be desirable to start new processes during the course of the simulation. For instance, if the length of the customer queue becomes rather large, the manager may decide to hire another barber. One solution to this would be to include the additional barber from the beginning, but to keep his process suspended by a suitable semaphore until required. This is a cumbersome solution, and if the number of new processes that might be required during a run of the simulation is not known with any certainty, and a reliable upper bound is large, there are more serious problems.

4. A further problem with parallel processing is the need to use semaphores. To some extent the simulation package procedures can deal with them, but not in all cases, and it is undesirable to have to inflict this additional burden on the simulator.

The truth of the matter is that the simulator does not want true parallel processing. Although a simulation model can be regarded as a set of interacting activities carried out in parallel, the events which change the state of the simulated system are regarded as taking place instantaneously. Indeed, if two events are to take place at the same instant of simulated time, the simulator may have a preference as to the actual order of execution, and will write his program accordingly.

If one wants to use the process control approach in Algol 68, and it has many attractions for simulation, the obvious way is to extend the language and use the concepts of quasi-parallel processing that have been developed in Simula. Research would have to be undertaken to determine a form of quasi-parallel that was consistent with the spirit of Algol 68, but there appears to be no reason in principle why this should not be done.

## Queues

There are, clearly, problems in providing a general list processing/queueing package that can be placed, once and for all, in a library prelude. Lindsey has discussed possible extensions to the language

to cater for such facilities, but it remains to be seen if these ideas can be developed to deal with rather more complicated operations than he illustrated, and at the same time to do this in a way which the simulator will find convenient.

One reasonably satisfactory way of providing list processing facilities depends on the simulator being able to create his own private library prelude. In (1), the simulator must declare the modes of all variables that may be in a list (apart from ints etc.), and then declare a new mode setmem as the union of these. The list processing procedures, which depend only on the mode setmem, can now be compiled to form a library prelude for a particular simulation, and generally there will be a goodly number of runs as the program is debugged and developed. In the barber shop example, one might have

```
begin
mode barber = struct(bool smoking, int haircuts),
     customer = struct(int call, arrive),
     setmem = union(ref barber, ref customer);
¢ the list processing procedures ¢
end
```

Provided that one has a good compiling system, this can be a fairly painless business. The cost on an ICL 1907 using the Algol 68-R system is about 10 seconds of mill time for a fairly large set of list processing procedures.

There are certain disadvantages with this approach, such as not being able to detect at compile time that a ref customer is being attached to a queue that is supposed to be reserved for ref barbers. However, when one takes an element from a list one has to use the conforms-to-and-becomes operator (::=), and so a check is done then. Use of a union is clearly inefficient in the use of space, but this can be minimised if all elements that may be members of lists are declared as int, real, bool, or ref ..., and as Lindsey has pointed out this is no great inconvenience.

An advantage of the approach is that it does allow for the declaration of procedures and operators that are reasonably straightforward to use, and at the same time are quite powerful. For instance, to create a new customer, and place a reference to him on a list called "waiting", one can write

```
( heap customer := (time,0) ) joins waiting
```

As a more complicated example, suppose that one wished to remove a reference to a barber in a list called "free barbers", making sure that he is a barber and is not smoking, and if there is a choice to take one who has completed the smallest number of haircuts so far. In (1), one could write, using proceduring to aid intelligibility,

```
setmem sm;  ref barber rb;
remove sm within free barbers
            satis ( rb::=sm | not smoking of rb | ¢ error ¢ )
            minim (haircuts of rb);
if none found then . . . fi;
¢ rb is now a reference to the required barber ¢
```

Practical experience, though limited, suggests that this approach is actually quite efficient when used in a simulation, though if one's program contained very little but list processing, it might be less so.

## General Comments

It is clear that Algol 68 is not ideally suited to simulation in the sense that one cannot build a set of general list processing procedures that can be placed in a library prelude, and one cannot produce a satisfactory process control executive. None the less, one can design a reasonably satisfactory package along the lines discussed earlier, and those who use Algol 68 for other aspects of their work should be able to write simulations without difficulty. Whether or not one should learn Algol 68 just in order to carry out simulations is a different matter, given the many other simulation languages available. Extensions to Algol 68 in this direction will therefore be viewed with considerable interest.

## Reference

1. D C S Shearn, Algosim: a Simulation Language based on Algol 68, Division of Economic Studies, University of Sheffield (1973)

M.R.Levinson.
Moscow,
Leninsky Prospect 14 Korp.7,
CEMI AS USSR.

{Editor's note -
This paper is gleaned from various letters which have passed between me and Mr Levinson since his paper on the same topic in AB36.4.2 (see also corrections thereto in AB37.1.1 and further proposals by C.H.Lindsey in AB37.4.2 and AB37.4.3).}

The library-prelude presented in AB36 allowed a process to wait for an inactive period of some fixed number of time intervals. It is also necessary for a process to be able to wait for the completion of some event happening at a future unforseen moment. For example, suppose that a barber has one smoke when his queue of customers is exhausted and then, if there are still no customers in the queue, goes to sleep until a customer appears. For this, I propose a new operator seize:

```
sema sim sema = level 0;
op seize = (sema a) void:
     (test:
          if down sim sema; level a > 0
          then down a; up sim sema
          else up sim sema; wait 1; go to test
          fi);
```

However, although this operation is useful in some cases, I did not find it applicable to an easy (for an ordianary user) solution of the problem about the smoking and sleeping barber. I got such a solution with the aid of a dyadic version of this operator:

```
op seize = (sema a, b) void:
     (test:
          if down sim sema; level a * level b > 0
          then down a; down b; up sim sema
          else up sim sema; wait 1; go to test
          fi);
     co both semaphores are seized if both are available; if
     either is unavailable, both are left up during the wait co
```

Now, in the smoking barbers program (AB36 version), you can declare sema waiting = level 0 (it should have been int waiting := 0 before), and replace all occurrences of waiting +:= 1 (-:= 1) by up (down) waiting . The process barber then becomes:

```
process barber = (int number) void:
     do ref customer client;
          if   down queue sema;
               ref customer (client := next please) :/=: nil
          then      next please := next of next please;
               down waiting;
               up queue sema
          else up queue sema;
co smoke co    wait poisson (smoke time);
co sleep co    queue sema seize waiting;
               client := next please;
               next please := next of next please
          fi;
          call of client := time;
          wait poisson (haircut time);
          print (...)
     od;
```

Now, if the barber is sleeping and a new customer enters, waiting is upped and, when next this sema is inspected after the wait in seize, the barber takes him from the queue and proceeds to cut his hair. This may happen in the same time interval as that in which the customer entered, or in the following one, according to the manner in which the new customer and barber processes are merged. However, in simulation with an integral time axis, one time period must be negligibly small as compared with the whole simulation time and so all collisions of simultaneous actions can be resolved by "distribution" on a number of time periods.

If the language were to be extended by the "Modals" proposal (AB37.4.3), then the seizeing could be integrated into the queue handling. Please make the following alterations to AB37 p29:

1. The length of the queue becomes a semaphore:
```
queue      # int length => sema length #
include+3  # length of q +:= 1 => up length of q #
remove+5   # length of q -:= 1 => down length of q #
initiate+2 # 1, 0 => 1, level 0 #
```
2. The use of sema of q is protected by sim sema:
```
include+1, remove+1
           # down sema of q =>
             down sim sema; down sema of q; up sim sema #
```
3. A new subroutine is added:
```
proc seize = (mode x, ref queue(x) q) ref x:
     (sema of q seize length of q;
          ref x object = first of q;
          first of q := next of first of q;
     up sema of q;
     object);
```
which differs from remove only in that it guarantees the selection of some element from the queue (perhaps at the score of waiting).

4. A new field appears in queue, being the procedure seize:
```
queue+1    # outproc => outproc, getproc #
initiate+2 # remove (x, q) =>
             remove (x, q), seize (x, q) #
```
5. With the inclusion of these alterations, the process barber, providing smoking and sleeping, has the form:
```
process barber = (int number) void:
     (ref customer client;
     do   client := outproc of waiting room;
          if (ref customer (client) :=: nil
          then
co smoke co    wait poisson (smoke time)
co sleep co    client := getproc of waiting room
          fi;
          call of client := time;
          wait poisson (haircut time);
          print (...)
     od);
```

# An interpretation for making references (in ALGOL 68)

by Harry Feldmann

University of Hamburg

Computing Reviews Category: 4.12

Key words and phrases:
Algol 68, to refer to, to assign to, slice, field selection, object, graphical interpretation.

Summary:

This paper gives a graphical interpretation for making references between objects which is both convenient in use and of high precision. The number of independent primitive concepts used in the interpretation has been minimized. One may consider it as an advantage for compilation and didactical aims or eventually as a loss of generality (in the future) that this interpretation-model makes use of the present computer-concept of "address and content of storage cell or cells" (R 2.1.3.2.a).

## 1. Graphical interpretation

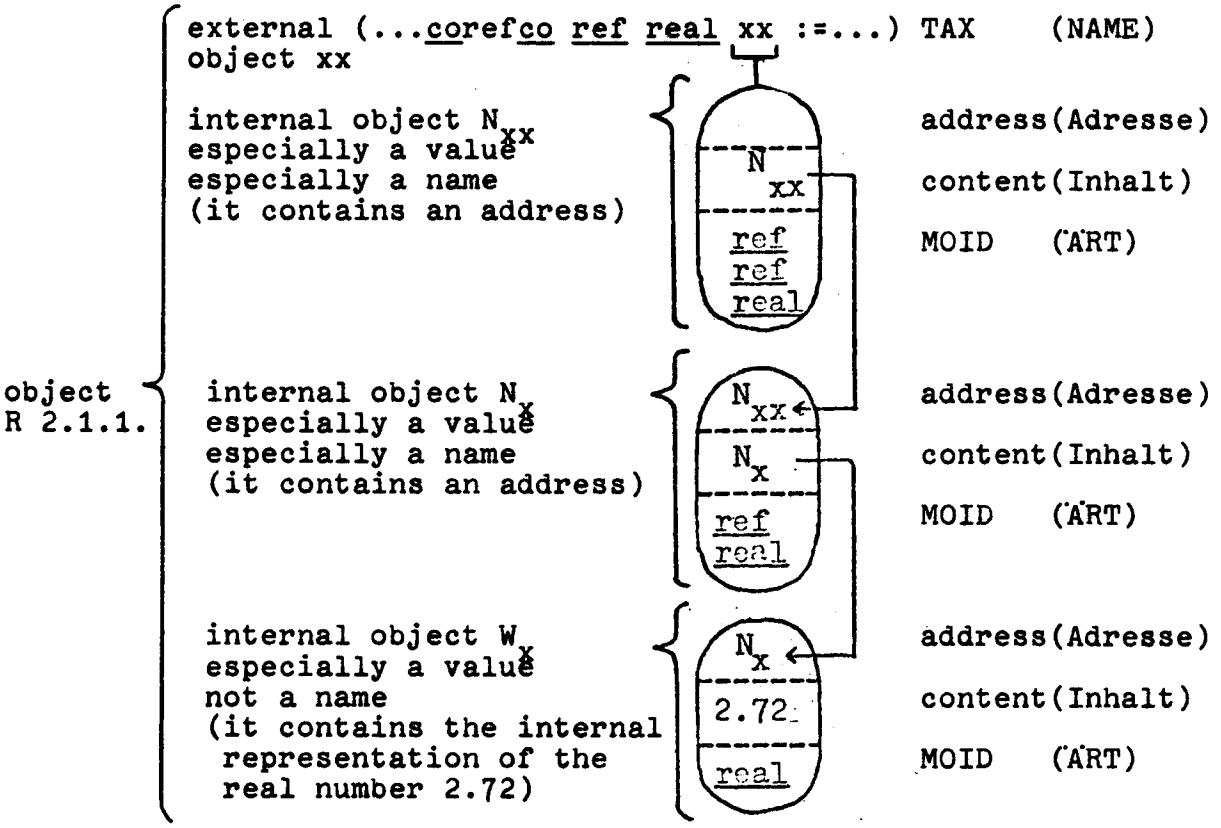
The "Revised Report" [1], cited "R", does not contain any graphical interpretation for (external and internal) "object"s, although it is allowed to use some (R 2.1.3.2.a). Every ALGOL 68 - compiler would give an (graphical representable) interpretation.

We choose a simple interpretation-model in which each internal object (R 2.1.1) is represented by a graphical object composed of two parts, the "address" and the "content" (and of a third part, the "MOID", which could be put together with the "content).

This model is slightly more detailed and more dependent to present machine-concepts than the comparable interpretation choosen in Lindsey, van der Meulen [2].

For better understanding we give some German translations in brackets used in Feldmann [3] .

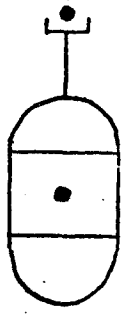
Let us explain the interpretation-model in an example:




The name $N_{xx}$ is newly created by the elaboration of the sample generator corefco ref real and is different from all other names (R 2.1.3.2.a).

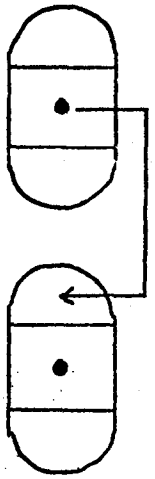
"Creation" is not always necessary for "ascription" (see below). The example real e = 2.72 (see part 2) shows, that e is ascribed to an internal object $W_e$ which is already existing (created for 2.72). See "identity declaration" (R 4.4.2.a) or others ("call" , "formula", "cast", "yield of assignation" etc.).

The name $N_{xx}$ is ascribed to the reference-to-reference-to-real-defining-indicator -with-letter-x-letter-x (R 4.8.2.a).
The name $N_{xx}$ is the yield of the reference-to-reference-to-real-applied-indicator-with-letter-x-letter-x (R 4.8.2.b).
The name $N_{xx}$ is accessed by the reference-to-reference-to-real-letter-x-letter-x (R 2.1.2.c).



"ascription" or "yield" or "access" represented by a line

The name $N_{xx}$ is a value which is made to refer to the value $N_x$ and the name $N_x$ is a value which is made to refer to the value $W_x$ (R 2.1.3.2.a). The mode of the name $N_{xx}$ is reference-to-reference-to-real and the mode of the value $N_x$ which is referred to by $N_{xx}$ is reference-to-real. The mode of the name $N_x$ is reference-to-real and the mode of the value $W_x$ which is referred to by $N_x$ is real (R 2.1.3.2.b). The name $N_{xx}$ (resp. $N_x$) refers to the value $N_x$ (resp. $W_x$). This relationship is made to hold when $N_{xx}$ (resp. $N_x$) is made to refer to $N_x$ (resp. $W_x$) and ceases to hold when $N_{xx}$ (resp. $N_x$) is made to refer to some other value (R 2.1.2.e).



"reference" represented by an arrow

## 2. Making references by assigning

We consider the semantic term "is assigned to" of the assignation (R 5.2.1.2.b) respectively of the variable-declaration

(R 4.4.2.b) in the case NONSTOWED (to which all other cases can be reduced):

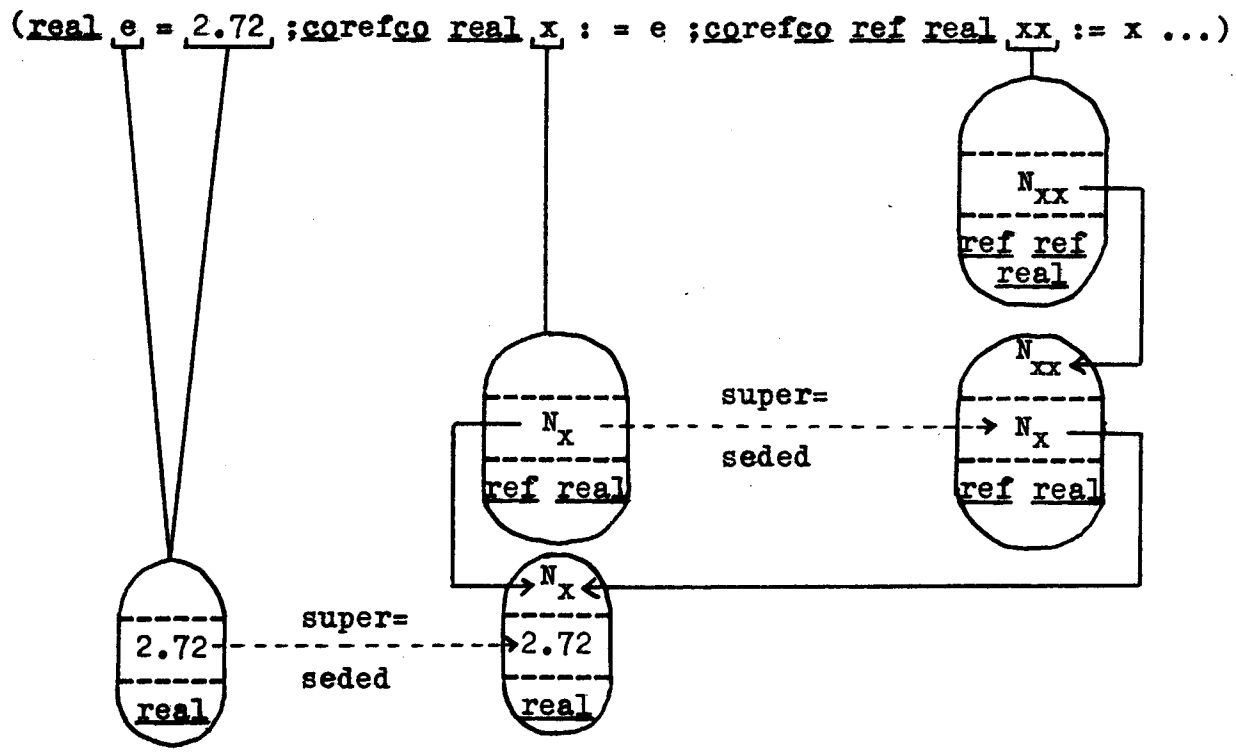
> "N and W are the yields of the destination (Verweisender) and the source (Verwiesener).
> N is made to refer to W " (R 5.2.1.b)

and interprete the semantic term "N is made to refer to W" as

> "The content of the internal object to which N refers is superseded by the content of W" .

"Superseding" which is not to be found in the Revised Report can be represented by a dotted arrow in our graphical model (if desired).

The following example shows that the so interpreted "assigning" makes "references between internal objects" only if the mode of the destination has at least two ref s, like corefco ref real xx := x , otherwise it makes only "unconnected copies", like corefco x := e .



(The author could give another interpretation of the semantic term "N is made to refer to W" as

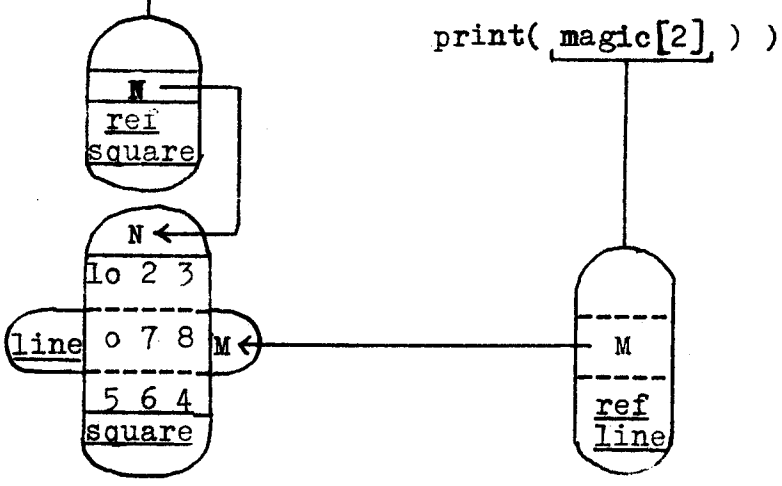
|| "The content of N is superseded by the address of W"

according to which "assigning" would always make a "reference between internal objects" including the case that the mode of the destination has only one ref ,
but this interpretation could violate the "new creation" or the "ascription or access or yield" of N . Surely it would lead into ambiguities concerning the identity relation, because then ( real x,y ; x:=y:=2.72 ; x:=:y ) would yield true.)

## 3. Making references by slicing and field-selection

Slicing can generate a new name (R 5.3.2.2.a). The name M generated by a trim T from a name N which refers to a multiple value V is a {fixed} name of the same scope as N, {not necessarily newly created} which refers to the multiple value W selected by T in V (R 2.1.3.4.j). In the following example there is made a "reference between the internal object M and W" by slicing :

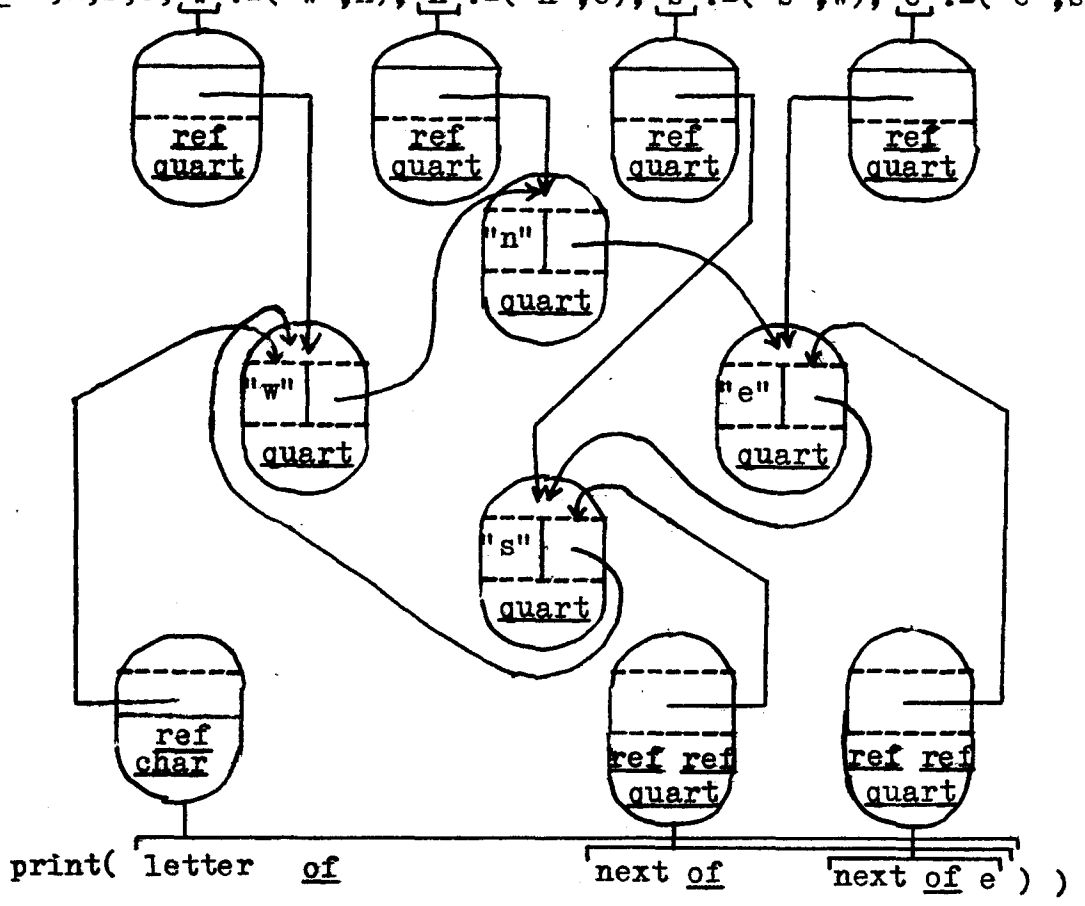
```
(co goethe faust 1 , kitchen of the witch co
mode square = [1:3,1:3] int , line = [1:3] int ;
coref co square magic := ((1o,2,3),(o,7,8),(5,6,4)) ;
                                    print( magic[2] ) )
```



In order to keep the graphical representation easily to be surveyed and to avoid redundances, information about descriptors is only given in the MODE of the internal object. Different graphical MODE-parts indicate different descriptors. Identical graphical content-subparts indicate the same subcontent without making copies.

Selection too can generate a new name (R 5.3.1.2.). The name M generated by a {field-selector} TAG from a name N which refers to a {multiple} value V each of whose elements is a structured value is a {fixed} name of the same scope as N, {not necessarily newly created} which refers to the multiple value selected by TAG in V (R 2.1.3.4.1). In the following example there are made three "references between internal objects" by field-selection:

```
(mode quart = struct( char letter , ref quart next );
quart w,n,s,e; w :=("w",n), n :=("n",e), s :=("s",w), e :=("e",s);
```



In order to keep the graphical representation easily to be surveyed and to avoid redundances information about the MODE s of the field-elements of internal objects is only given in the MODE of the whole internal object. The "reference between an internal object and a field-element E of another internal object O " is graphical represented by an arrow running through the address-part of O to the border adjacent to the content-subpart of O belonging to E. Identical graphical content-subparts indicate the same subcontent without making copies.

## Literature

[1] A. van Wijngaarden et al.: "Revised Report on the Algorithmic Language ALGOL 68" , to appear in Acta Informatica.

[2] C.H. Lindsey, S.G. van der Meulen: "Informal Introduction to ALGOL 68" , North Holland Publishing Company, Paperback Edition, 1973.

[3] H. Feldmann: "Einführung in ALGOL 68" Lecture script, University of Hamburg, complete edition to appear 1975.

The following are corrections to Technical Report TR74-3, the Revised Report on the Algorithmic Language ALGOL 68, issued in March 1974 as a supplement to ALGOL Bulletin 36. They are supplemental to those published as ERRATA-2 in ALGOL Bulletin 37, and are to be applied after them.

Category A (significant errors)

p87 6.6.2.a+3:+5
```
# If ... ♣'ROWS1'•. =>
Case A: ♣'REFETY'• is ♣'EMPTY'•:
▪   ▪W• is the multiple value "built" {b} from ▪V•
for ♣'ROWS1'•;
Case B: ♣'REFETY'• is ♣'REF to'•:
   If ▪V• is nil,
   then ▪W• is a nil name;
   otherwise, ▪W• is the name "built" {c} from ▪V•
for ♣'ROWS1'•. #
```

p88 6.6.2.b
```
# a multiple => a #
```

p106 9.2.1.a
```
# A341b,h => A341b,h,A348a,b,c,A349a,A34Ab #
```

p111 9.4.1.f+4,+5
```
# 133d => 133d,A348b,A34Ab #
```

p149 10.3.1.6.j+3:+5
```
# ELSE ... set read mood (f); =>
ELSE
  REF BOOL (read mood OF f) := ¬ put possible (f);
  REF BOOL (write mood OF f) := ¬ get possible (f);
  REF BOOL (char mood OF f) := ¬ bin possible (f);
  REF BOOL (bin mood OF f) := FALSE; #
```

p151 10.3.2.1.c+19
```
# ELSE fixed (v, width, after - 1) =>
ELIF after > 0
THEN fixed (v, width, after - 1)
ELSE ABS width * errorchar #
```

p152 10.3.2.1.e-7
```
# exp  ≥ 0 => exp > 0 #
```

p165 10.3.4.1.1.i
```
# A348a,b => A348c #
```

p176 10.3.4.8.1.a+2
```
# literal{•A341i♣} list brief pack. =>
praglit{•c♣} list brief pack,
pragment{•92a♣} sequence option. #
```

p177 10.3.4.8.1.b+2,+3
```
# literal{•A341i♣} => praglit{•c♣} #
```

b+4
```
# .• => , pragment{•92a♣} sequence option.•
c)♣NEST praglit{•a,b♣} :
pragment{•92a♣} sequence option,
NEST literal{•A341i♣}.• #
```

p178 10.3.4.9.1.a+2
```
# . => ,  pragment{•92a♣} sequence option. #
```

p179 10.3.4.10.1.a+2
```
# brief pack => #
```

b
```
# : => :  brief begin{•94f•} token, #
```

b+1
```
# . => , brief end{•94f•} token,
pragment{•92a♣} sequence option. #
```

p180 10.3.4.10.2.+8
```
# pack- => #
```

p206 12.1."built (the name"
```
# the name of a multiple value => a name #
```

12.1."built (the value"
```
# value built from a multiple value =>
multiple value built from a value #
```

p213 12.2."power-of-ten"
```
# 8.1.2.1.i => 8.1.2.1.i
♣praglit•  10.3.4.8.1.c #
```

Category B (clarifications)

p4 0.2.3.+3
```
# after => before #
```

p5 0.3.+2:+5
```
# For a more ... a revised edition =>
The revised language will be described in a new
edition #
```

p6 0.3.2.+1
```
# enquiry => enquiry-clause #
```

p16 1.1.3.3.a+1:+2
```
# or "Metasyntax" =>
, "Metasyntax" or "Metaproduction rules" #
```

p20 1.1.4.2.a+4
```
# must designate => designates #
```

p21 1.1.4.2.b(ii)+7
```
# or 9.1.1.f => #
```

c+9
```
# constituent (d) => #
```

p32 2.1.3.1.c+3
```
# ▪n• => ▪-n• #
```

p33 2.1.3.1.h+3
```
# or ♣formal-void =>
,  ♣void-identity-declarations• or ♣void #
```

p34 2.1.3.2.a+2
```
# . =>
; moreover, for each mode beginning with
♣'reference to'•, there is exactly one nil name of
that mode. #
```

a+7
```
# all other names =>
all names already in existence #
```

p42 2.1.5.d+4
```
# of them => of the elaboration of them #
```

p43 2.2.2.b+4
```
# except => except certain modes like #
```

b+10:+15
```
# possibilities. It <TEXT1> information. It
<TEXT2> finite. =>
possibilities. It <TEXT2> finite. It <TEXT1>
information.#
```

```
p55   3.4.2.b.Case C+2
                    # : => {; if there exists more than one such
                    constituent •specification•, it is not defined
                    which one is chosen as ¤D•}: #

p57   3.5.2."This equivalence"+0:+1
                    # holds, of course ... identify =>
                    might not hold, of course, if the •loop-clause•
                    contains •local-generators•, or if some of the
                    •operators• above do not identify #

p58   4.1.2.a       # a) => #

      b:b+2         # b) ... definitions. => #

p64   4.6.1.Examples.a
                   ¤# REAL => [1 : n] REAL #

p65   4.6.2.b.Case A+4
                   # •field-selector• =>
                   •defining-field-selector• #

p66   4.6.2.d-5     # direct => visible direct #

p67   4.8.1.a+3,c+2 # PROPSETY1 => PROPSETY1 PROPSETY2 #

p75   5.3.2.2.-5    # •trimmers•. =>
                    •trimmers• or •revised-lower-bound-options•. #

p83   6.1.1.-2      # 9.1.k => 9.1.f #

p84   6.2.-21       # enquirys => enquiry-clauses #

p87   6.5.2.Case B+2 # the structured => a structured #

      6.6.2.a+2     # direct descendent => •STRONG-FORM• #

p93   7.2.2.c+7     # identifies {b} a •defining-indicator• not =>
                    does not identify {b} a •defining-indicator• #

p95   7.3.1.q+7     # MODE1 MODE2 => MOID1 MOID2 #

p99   8.1.0.1.Examples
                    # Examples => Example #

      Examples.a    # • 1.23e-4 => #•

p101  8.1.3.2.+2    # token => symbol #    twice

p102  8.1.4.2.-1    #  •  è  •  ⊥ => #

p104  8.2.2.c+2     # 9.3 => 9.3.b #

      8.3.2.+6      # symbol => •symbol• of the •string• #

p110  9.4.1.b."times ten to the"
                    # ,814c => #

      c."plus i times"
                    # 814c, => #

p111  9.4.1.f-1     # 942e => 942E #
                    {incorrectly specified in ERRATA-2}

      f."open symbol","close symbol"
                    # ,A348b => #

p116  10.1.2.c      # , if any, => #

p117  10.1.3.Step 3 # made => modified or made #

      Step 4+3      # or ¤L ¤BITS => , ¤L ¤BITS• or ¤L ¤BYTES• #

p120  10.2.1.t+1    # b,e,f => b,c,d,e,f #

p124  10.2.3.4.p   ¤# can be lengthened to => is widenable to #

p128  10.2.3.10.i+1 ¤# (INT n ... ]; =>
                    (INT n = (INT la = ⌜ a [@ 1]; la < 0 | 0 | la),
                       n = (INT lb = ⌜ b [@ 1]; lb < 0 | 0 | lb); #

p130  10.2.3.11.q+1,r+1
                    # (a + b) => a + b #

      u+1           # (a * b) => a * b #

p131  10.2.4.e+2    # ≥ 1; => ≥ 1 #
p143  10.3.1.4.e+6  # OF f => OF file #

p157  10.3.3.1.a+18 # ◀(UNION (NUMBER, ¤L ¤COMPL) =>
                    (UNION (NUMBER, ◀¤L ¤COMPL▶)  #

      a+21,+22,+23  # (¤L =>¤ ◀(¤L #

      a+21,+22      # ) , => )▶ , #

      a+24          # ) => )▶ #

      a+37          # ▶ => #

p160  10.3.3.2.a-7  # end• =>  end•, #

p172  10.3.4.1.2.f.Other cases+1
                    # •symbol• => constituent •symbol• #

p177  10.3.4.8.b+2  # open => brief begin #
      b+4           # close => brief end #

      10.3.4.8.1.Examples.b
                    # } =>
                    c) ¤"mon"•} #

      10.3.4.8.1.aa+5,bb+4
                    # •literal-list => •praglit-list #

      dd+6          # •literal => constituent •literal #

p178  10.3.4.8.1.ee+3 # •literal => constituent •literal #

      10.3.4.8.2.+4 # •literal-list => •praglit-list #

p180  10.3.5.b+7    # forp => forp OF f #
```

```
p184    10.3.5.1."int pattern"+2
                        # 1 +:= rep => (rep  ≥ 0 | 1 +:= rep) #

p185    10.3.5.1."edit 1 int"-1, "charcount"+1
                        # 1 +:= rep => (rep  ≥ 0 | 1 +:= rep) #

        "edit 1 int"+3  # s) => s) ¢ 1 = 0 #

        "edit 1 real"+14
                        # s) => s) ¢ a + b = 0 #

p187    10.3.5.1.a."gpattern"+8
                        # ))) => ) | () )) #

p200    11.7.+3         # LWB p => LWB p + 1 #

p204    11.12.+17       # PROC (INT) => PROC (INT) VOID #

p208    12.1."logical end"
                        # 10.3.1.1. => 10.3.1.1.aa #

p209    12.1."newly created"
                        # 2.1.3.2.a => 2.1.3.2.a
                        nil 2.1.3.2.a #

p211    12.1."weak"     # 6.1.1 => 6.1.1
                        {well formed} 7.4 #

p223    12.5."BITS"+1,"BYTES"+1
                        # SITHETY => field SITHETY #

p225    12.5."SAFE"     # MODE1 MODE2 => MOID1 MOID2 #

Throughout the entire Revised Report
                        # ¤nil¤ => nil #

Category C (minor misprints)

p viii "The original"+1
                        # whole-hearted => wholehearted #

p ix    "[1]"-3         # criticisms => criticism #

        "[5]"           # on aims. => on aims, #

p7      0.3.7.+6        # of course => of course, #

p10     1.1.1.b+3       # 1.1.3.4.d => 1.1.3.4 #

p12     1.1.3.1.e       # is either => either is #

p16     1.1.3.2.i+5     # shown => # #

p17     1.1.3.3.c-1     # (1.2.1.D) => (1.2.1.D)} #

p19     1.1.4.-9        # omitted => omitted, #

        1.1.4.1.+15     # ¤"QUALITY"¤ => "¤QUALITY¤" #

p21     1.1.4.2.c-5     # 4.8.1.e => 4.8.1.a #

p31     2.1.1.3.b+4     # in its => its #

        2.1.2.-4        # 5.2.3.2 => 5.2.3.2.b #

        2.1.2.a+2       # "transitive", => "transitive"; #

p35     2.1.3.4.b+2     # n¤ is => n¤, is #

p40     2.1.4.2.e+3     # which, => which #

p42     2.1.5.d+17      # ;"} => ;".} #

p64     4.6.1.s+1       # ¤47f¤ => ¤47f¤# 

p74     5.3.1.2.+3      # not be => be not #

p76     5.4.-15         # 2.1.3.4.g => 2.1.3.4.j #

p79     5.4.3.2.-2      # 10.3.1.1.b => 10.3.3.1.a #

p84     6.2.-20         # x1 [i]¤ and =>  x1 [i]¤, and #

        6.2.-13         # 6.4.1 => 3.2.1.e #

p91     7.2.-26         # mode(s)could => mode(s) could #

p102    8.1.4.1.d+4     # 'quote symbol' => ¤'quote symbol'¤ #

p103    8.2.1.g+1       # one{¤94b¤} symbol => one symbol{¤94b¤} #

p104    8.2.2.c+1       # ¤ digit => ¤ -digit #

p107    9.3.a+7         # chapter => Chapter #

p109    9.4.c+6         # 1.1.5.e => 1.1.5.b #

p118    10.1.3.Step 5-6 # a ¤and also =>¤ an ¤and-also # 

        10.2.-3         # Chapter => chapter #
                        {correction to a replacement text from ERRATA-2}

p149    10.3.1.6.i+10   # undefined) => undefined); #

        i+11            # ; => # 

p150    10.3.1.6.k+11   # ; => # 

p155    10.3.2.3.c+2    # not be a name which is =>
                        {if it is a name} be not #

        d+2             # united => united, #

p159    10.3.3.2.aa     # first => first, #

p162    10.3.4.+2       # cf => cf. #
```

```
p165    10.3.4.1.1.Examples.b+1
                        # si"+ => si + #

p166    10.3.4.1.1.aa+3  # cf => cf. #

        ee+2             # An => an #

p174    10.3.4.4.1.Examples
                        # Examples => Example #

        10.3.4.5.1.Examples
                        # Examples => Example #

p175    10.3.4.6.1.cc+3  # ◆S• => ■S• #

        10.3.4.6.1.Examples
                        # Examples => Example #

p176    10.3.4.7.1.bb+4  # radix-two => radix-two- #

        cc+6             # ◆B• => ■B• #
                         # □I => ■I #•

p177    10.3.4.8.1.cc+1  # then => them #

p178    10.3.4.8.2.+7    # The => the #

        10.3.4.9.Examples
                        # Examples => Example #

p193    10.3.6.+2,+14    # cf => cf. #

        10.3.6.+6        # reinput => re-input #

p196    10.4.1.a         # s) ; => s) ;• #

p200    11.8-2           # call => ◆call• #

p203    11.11."list"+10  # 180 => 180 #
```

## QUESTIONNAIRE TO IMPLEMENTERS ON THE PROPOSED REVISION TO ALGOL 60

| Official Name of the Implementation (include Mk. nos, etc., as appropriate) | Computer or Computer Family on which it runs | |
|---|---|---|
| | Manufacturer | Model No. |
| | | |

Name of Company or organisation responsible for this implementation:

If the Company is responsible for more than one implementation of ALGOL 60, please fill in a separate questionnaire for each one.

| Name of person making this Report | Are you making this Report as: |
|---|---|
| | 1. The person responsible for the implementation within the company mentioned above? ☐ |
| | 2. An interested user of that implementation? ☐ |
| | 3. Other? (please specify) ☐ |

Address for Correspondence

The answers to the questions overleaf are intended to help the IFIP Working Group 2.1 to decide whether to press ahead with the proposed revision and, if so, which specific changes to include. They are not intended to bind your company to any particular viewpoint, or to commit it to implementing any change that might be made.

Completed questionnaire should be returned to:

B.A. Wichmann
Division of Numerical Analysis & Computing
Department of Trade and Industry
National Physical Laboratory
Teddington
Middlesex
TW11 OLW

Each of the following four questions should be answered, in the appropriate column, for each of the changes proposed.

1. Does your implementation already include this feature (whether by design or by accident)? Possible answers Y, N or ? (Please explain if ?).

2. Would the implementation be invalidated by the change (or further invalidated if already invalid by the present Revised Report)? Possible answers Y, N or ? (explain ?).

3. Do you approve of the proposed change (irrespective of whether your implementation does, or may in the future, include it). Possible answers Y, N or ? (explain ?).

4. If the proposed change were made official, is it probable that your implementation would be brought into line. Possible answers Y, N, ?, - or blank. Please answer "-" if you answered "Y" to question 1. "Blank" implies that you are an interested user, rather than a person qualified by an official connection with the implementation.

| Change | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|
| 1. static own variables | | | | |
| 2. only fixed bounds for own arrays | | | | |
| 3. own variables initialized to zero or false | | | | |
| 4. step expression of <for statement> to be evaluated only once per cycle | | | | |
| 5. controlled variable of <for statement> not to be subscripted | | | | |
| 6. controlled variable to remain defined on exit | | | | |
| 7. comments to consist of characters rather than ALGOL basic symbols | | | | |
| 8. strings to consist of characters rather than ALGOL basic symbols | | | | |
| 9. no integer labels | | | | |
| 10. <integer> ↑ <negative integer> undefined | | | | |
| 11. goto undefined switch designator undefined | | | | |
| 12. <specification part> for all <formal parameter>s | | | | |
| 13. environment enquiries maxreal, minreal, maxint and epsilon | | | | |
| 14. IFIP input/output procedures insymbol, outsymbol, etc. | | | | |
| 15. additional standard procedures outterminator, fault and stop | | | | |

Please indicate any further comments or suggestions.