ISSN 0084-6198

Algol Bulletin no. 39

FEBRUARY 1976

CONTENTS

Important notice to LIBRARIANS

If this copy of the ALGOL BULLETIN is to be placed in a library, please first detach pages 33-38 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 and AB38). Better still, modify your copy in accordance with all the 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).

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, AB37 and AB38 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.

AB39.0 EDITOR'S NOTES

It is over a year since our last issue, and my apologies for that. However, you will see that the bulk of this issue is taken up with matters arising from the last Working Group meeting and with other official business. Whilst the publication of such material is an important function of the ALGOL Bulletin, it is not its only one and it is our desire to publish contributed papers, letters, opinions, reports, algorithms, etc., etc. on all aspects of Programming Languages (both existing and projected). During the past year, such contributions have been conspicuously absent. The answer is obvious and lies entirely in your hands, dear readers.

That Report, at last!

The Revised ALGOL 68 Report was published in December in Acta Informatica Vol. 5, parts 1, 2 and 3 (see announcement in this issue concerning availability of reprints). It is due to be published also in SIGPLAN notices and translations into Russian and into German are underway. It is also probable that an edition in Braille will be produced by the Mathematisch Centrum, Amsterdam.

WG2.1 meeting, Munich, August 1975

This departed from the usual pattern of Working Group meetings insofar as the bulk of the time was taken up with an Informal Conference with presented papers. The aim was to survey the whole field of Algorithmic Languages in order to identify those areas in which the Working Group could most usefully employ its talents in the future. The same format is likely to be adopted at the next meeting, which is scheduled to be held in France in August 1976. The papers given at Munich, together with the discussions, are being edited by Steve Schuman and will be distributed, courtesy of I.R.I.A., as a Supplement to this Bulletin.

As well as looking forwards to its future responsibilities, the Working Group also took some important decisions with regard to its past and present work:

Modified ALGOL 60

A Revised version of the "Commentary on the ALGOL 60 Revised Report" (AB38.3.1), by R.M. De Morgan, I.D. Hill and B.A. Wichman, was presented to the meeting, and it was agreed that it should be published as an IFIP document. See the article by Brian Wichman in this issue for further details.

Taken together with the Revised Report, the new Supplement defines the language "Modified ALGOL 60" (which I suspect will soon become abbreviated to "ALGOL 60 M"). It has been offerred for publication to the three journals

which published the original Revised Report. I hope to publish the complete Report obtained by elaborating the Supplement in a future issue of the ALGOL Bulletin.

ALGOL 68 Sublanguage

A draft specification for an ALGOL 68 Sublanguage (see AB37.4.4 for an earlier draft - fortunately the present version is much more readable than that one) was presented by P.G. Hibbard and, after a few minor changes, was released for publication as an IFIP document. Peter Hibbard is to prepare it for publication, but his recent removal to Carnegie-Mellon University has introduced some delay into the schedule.

The Sublanguage is intended for easy implementation on minicomputers. Implementations corresponding to the earlier draft exist on a Modular 1 at Liverpool and on a 370 (some mini!) at Durham. The latest version is being implemented on a PDP11 at Carnegie-Mellon.

I guess this Sublanguage is going to become known as "ALGOL 68 S".

Hardware Representation

A draft specification for a Standard Hardware Representation for ALGOL 68, prepared under the supervision of the ALGOL 68 Support Subcommittee, was presented by H.J. Boom and W.J. Hansen. This aims to facilitate portability of ALGOL 68 source texts by fixing the stropping conventions and by adhering to a minimal set of "worthy" characters. This also was released for publication as an IFIP document, and it will appear in the next issue of the ALGOL Bulletin.

Any actual implementer who is in urgent and genuine need of advance information on this topic should write to H. Boom, Mathematisch Centrum, 2e Boerhaavestraat 49, Amsterdam for a copy of the latest Draft.

Subcommittee on ALGOL 68 Support

This subcommittee met for three days in Munich, prior to the main Working Group meeting. Topics discussed included partial parametrization, hardware representation (see above), modules and pre-compilation, and modals.

On partial parametrization, they approved a document which appears in this issue (AB39.3.1). Note that this has only a semi-official status. It is not defined as part of ALGOL 68, but is offerred as a suggestion to implementers who require a language feature of this nature.

On modules and pre-compilation, there were two schools of thought - "Top down" and "Bottom up". Two papers in this issue (AB39.4.1 and 39.4.2) present the two sides of the argument. A working party has been appointed to examine the pros and cons further, and to see whether a system embodying both features would be appropriate.

AB39.1 Announcements

AB39.1.1 The Revised Report on the Algorithmic Language ALGOL 68

Reprints of the Acta Informatica edition will be available from Springer-Varlag sometime in March, at a price of Dm 24. A certain number will also be available, hopefully during February, from the Mathematisch Centrum, 2e Boerhaavestraat 49, Amsterdam, at HF1 25.

AB39.1.2 Conference Proceedings : 1974 International Conference on ALGOL 68

University of Manitoba, Winnipeg June 1974, Editor : Peter R. King. Copies of these proceedings may be obtained from Utilitas Mathematica Publishing Inc., P.O. Box 7, University Centre, University of Manitoba, Winnipeg, Manitoba, Canada R3T 2N2. The price is $ 14.00 (Canadian). The 320 page volume includes invited addresses by I. Currie, P.G. Hibbard and R. Uzgalis as well as nineteen other contributed papers and a written record of the discussion periods.

AB39.1.3 Conference Proceedings : 1975 International Conference on ALGOL 68

Oklahoma State University, Stillwater, June 1975.
The Conference was well-attended by about sixty registrants, and had about thirty presentations during the three days. Copies of the proceedings may be obtained from G.E. Hedrick, Oklahoma State University, Department of Computing and Information Science, Stillwater, Oklahoma 74074, U.S.A. The price is U.S. $ 12.00.

AB39.1.4 Fourth International Conference on the Implementation and Design of Algorithmic Languages

To be held June 14-16, 1976 at New York University, Courant Institute of Mathematical Sciences. This Conference will be held under the auspices of the Algol Informal Information Interchange. Previous conferences have concentrated on the ALGOL 68 language, but many related areas have been covered and the forthcoming conference will have a broader scope covering algorithmic languages in general.

In addition to presentation of invited and submitted papers on aspects of this area, there will be a series of tutorial sessions covering topics including the following: The PL/I Basis/I definition, The SETL language. Inquiries may be addressed to the Program Committee Chairman: Robert B.K. Dewar, New York University, Courant Institute of Mathematical Sciences, 251 Mercer Street, New York 10012, U.S.A. Abstracts of papers intended for inclusion in the conference should be sent to the above address no later than April 1, 1976. Notices of acceptance will be mailed by May 1st, 1976. There will be a registration fee of $ 35.00.

AB39.1.5 A68 - III (Informal Information Interchange)

There are currently about 120 members of this organisation, many of them being active implementers. Currently, a questionnaire is being circulated to establish the progress and usage of the various implementations under way. An annual conference is organised (see separate announcement for details). A computerised bibliography is maintained and a Repository of ALGOL 68 related papers and technical reports is kept in the Computer Science archives at UCLA (photocopies available at about $ 0.05 per page). Items for inclusion in the bibliography and spare copies of Reports etc. for depositing in the Repository are always welcome.

Further details from: R. Uzgalis, Computer Science Department, School of Engineering and Applied Sciences, U.C.L.A., Los Angeles, California 90024, U.S.A.

AB39.1.6 The Progressive Construction of Mode-Trees in ALGOL 68

Ph.D. Thesis by G.S. Hodgson, University of Manchester, England. Jan. 1975, 174p. Microfiche copy available from C.H. Lindsey, Department of Computer Science, University of Manchester, Manchester M13 9PL, England, £0.80 (or $ 2.00 - dollar bills preferred to cheques).

"This thesis describes the progressive construction of mode-trees Representing ALGOL 68 modes. Intermediate Incomplete Representations are permitted until the full Representations can be determined. The influence of coercion on the method of Representation of modes is indicated.

"We describe an Equivalence Algorithm for ensuring that such Representations (or any parts of such Representations) are Unique. This is essential for the subsequent comparison of modes to be straightforward. Also given is a predefined Internal-Ordering for the component modes of a union. Separately compiled segments or programs may then be handled simply."

Also, still available, "The Transport Section of the Revised ALGOL 68 Report" by R.G. Fisker (see AB37.1.3), £0.40 (or $ 1.00).

AB39.3.1 Specification of partial parametrization proposal.

C.H.Lindsey (university of Manchester)

The following specification has been released by the IFIP Working Group 2.1 Standing Subcommittee on ALGOL 68 Support, with the authorization of the Working Group.

This proposal has been scrutinized to ensure that
a) it is strictly upwards-compatible with ALGOL 68,
b) it is consistent with the philosophy and orthogonal framework of that language, and
c) it fills a clearly discernible gap in the expressive power of that language.

In releasing this extension, the intention is to encourage implementers experimenting with features similar to those described below to use the formulation here given, so as to avoid proliferation of dialects.

{{Although routines are values in ALGOL 68, and can therefore be yielded by other routines, the practical usefulness of this facility is limited by scope restrictions. Consider:

```
proc f = (real x) proc (real) real: (real y) real: x + y;
proc (real) real g := f (3);
x := g (4)
```

This attempts to assign to g the routine "add 3". It does not work because the body of the routine is still fundamentally the routine-text (real y) real : x + y which expects to find the value x (i.e. 3) on the stack in the form of an actual-parameter of f, and by this time f is finished with and its stack level has disappeared. The problem arises whenever a routine-text uses identifiers declared globally to itself and the limitation is expressed in the Report by making the scope of a routine dependent on its necessary environ (7.2.2.c). Here is an attempt at functional composition which fails to work for the same reason:

```
proc compose = (proc (real) real f, g) proc (real) real:
  (real x) real: f (g (x));
proc (real) real sex = compose (sqrt, exp)
```

Clearly, if the restriction is to be lifted, a routine value has to have associated with it copies of these global values. Unfortunately, their number is in general indeterminable at compile time, and so the implementation of such values must be similar to that of multiple values referred to by flexible names (2.1.3.4.f) requiring, in most implementations, the use of the heap.

In this variant, all the intended global values appear to the routine-text as its own formal-parameters. At each call, some or all of these parameters are provided with actual values, resulting in a routine with that number of parameters fewer. Ultimately (possibly after several calls) a routine without parameters is obtained and, if the context so demands, deproceduring can now take place. Thus, all calls in the original language turn out to be parametrizations followed by immediate deproceduring, but their effect is the same. Here are some examples:

```
1)  proc f = (real x, y) real: x + y;
    proc (real) real g := f (3, );
    x := g (4)  # or x := f (3, ) (4) #

2)  proc compose = (proc (real) real f, g, real x) real: f (g (x));
    proc (real) real sex = compose (sqrt, exp, )

3)  op ↑ = (proc (real) real a, int b) proc (real) real:
      ((proc (real) real a, int b, real p) real:
         (real x := 1; to b do x *:= a(p) od; x)) (a, b, );
    real theta; print ((cos↑2)(theta) + (sin↑2)(theta)) }}
```

{{A routine now includes an extra locale.}}

2.1.3.5. Routines

a) A "routine" is a value composed of a routine-text {5.4.1.1.a,b}, an environ {2.1.1.1.c} and a locale {2.1.1.1.b}. {The locale corresponds to a 'DECSETY' reflecting the formal-parameters, if any, of the routine-text.}
b) The mode of a routine is some 'PROCEDURE'.
c) The scope of a routine is the newest of the scopes of its environ and of the values, if any, accessed {2.1.2.c} inside its locale.

{{A routine-text yields the new style of routine.}}

5.4.1.2. Semantics

The yield of a routine-text T, in an environ E, is the routine composed of
(i) T,
(ii) the environ necessary for {7.2.2.c} T in E, and
(iii) a locale corresponding to 'DECS2' if T has a declarative-defining-new-DECS2-brief-pack, and to 'EMPTY' otherwise.

{{Most of the remaining changes to the Report needed to incorporate this facility are in section 5.4.3 (calls).}}

5.4.3. Calls (with partial parametrization)

{A call is used to provide actual-parameters to match some or all of the formal-parameters of a routine. It yields a routine with correspondingly fewer formal-parameters or with none at all, in which case the yield is usually subject to deproceduring (6.3). Examples:

```
    y := sin (x)  .
    PROC REAL ncossini = (p | ncos | nsin) (i)  .
    print ((set char number ( , 5), x)) .}
```

5.4.3.1 Syntax

A) PARAMSETY :: PARAMETERS ; EMPTY.

```
a)  procedure yielding MOID NEST call{5D} :
      meek procedure with PARAMETERS1 yielding MOID NEST PRIMARY{5D},
        actual NEST PARAMETERS1 leaving EMPTY{c,d,e} brief pack.
b)  procedure with PARAMETERS2 yielding MOID NEST call{5D} :
      meek procedure with PARAMETERS1 yielding MOID NEST PRIMARY{5D},
        actual NEST PARAMETERS1 leaving PARAMETERS2{c,d,e,f}
         brief pack.
c)  actual NEST PARAMETER PARAMETERS leaving
         PARAMSETY1 PARAMSETY2{a,b,c} :
      actual NEST PARAMETER leaving PARAMSETY1{d,e},
        and also{94f} token,
        actual NEST PARAMETERS leaving PARAMSETY2{c,d,e}.
d)  actual NEST MODE parameter leaving EMPTY{a,b,c} :
      strong MODE NEST unit{32d}.
e)  actual NEST PARAMETER leaving PARAMETER{a,b,c} : EMPTY.

f)* actual MODE parameter :
      actual NEST MODE parameter leaving EMPTY{d}.
g)* dummy parameter :
      actual NEST PARAMETER leaving PARAMETER{e}.
```

{Examples:

```
a)  set char number (stand out, 5)
b)  set char number ( , 5)
c)   , 5
d)  5 }
```

5.4.3.2. Semantics

a1) The yield W of a call C, in an environ E, is determined as follows:
. let R {a routine} and V1, ... , Vn be the {collateral} yields of the PRIMARY of C, in E, and of the constituent actual- and dummy-parameters of C, in an environ E1 established {locally, see 3.2.2.b} around E, where the yield of a dummy-parameter is "absent";
. W is {the routine which is} the yield of the "parametrization" {a2} of R with v1, ... , Vn;
. except where C is the constituent call of a deprocedured-to-MOID-call {6.3.1.a}, it is required that W be not newer in scope than E {thus, proc (char, string) bool cs = char in string (, loc int,) is undefined but q := char in string ("A", loc int, s) is not}.

a2) The yield W of the "parametrization" of a routine R0 with values V1, ... , Vn is determined as follows:
. let T0, E0 and L0 be, respectively, the routine-text, the environ and the locale of R0, and let L0 correspond {2.1.1.1.b} to some 'DECS0';
. let L1 be a new locale corresponding to 'DECS0', and let the value, if any, accessed by any 'DEC0' inside L0 be accessed also by that 'DEC0' inside L1;
. let 'DECS1' be a sequence composed of all those 'DEC0's enveloped by 'DECS0' which have not {yet} been made to access values inside L1, taken in their order within 'DECS0';
For i = 1, ... , n,
  If Vi is not absent {see a1},
  then the i-th 'DEC1' enveloped by 'DECS1' is made to access Vi
    inside L1;
  {otherwise, the i-th 'DEC1' still does not access anything;}
. W is the routine composed of T0, E0 and L1.

{A routine may be parametrized in several stages. Upon each occasion the yields of the new actual-parameters are made to be accessed inside its locale and the scope of the routine becomes the newest of its original scope and the scopes of those yields.}

b) The yield W of the "calling" of a routine R0 in an environ E1 {see 5.4.2.2 and 6.3.2} is determined as follows:
• let T0, E0 and L0 be, respectively, the routine-text, the environ and the locale of R0;
• let E2 be a {newly established} environ, newer in scope than E1, composed of E0 and L0 {E2 is local};
• W is the yield, in E2, of the unit of T0.

{Consider the following serial-clause:

```
PROC samelson = (INT n, PROC (INT) REAL f) REAL :
  BEGIN LONG REAL s := LONG 0;
    FOR i TO n DO s +:= LENG f (i) ↑ 2 OD;
    SHORTEN long sqrt (s)
  END;
y := samelson (m, (INT j) REAL : x1 [j]).
```

In that context, the last deprocedured-to-real-call has the same effect as the deprocedured-to-real-routine-text in:

```
y := REAL : (
       INT n = m, PROC (INT) REAL f = (INT j) REAL : x1 [j];
       BEGIN LONG REAL s := LONG 0;
         FOR i TO n DO s +:= LENG f (i) ↑ 2 OD;
         SHORTEN long sqrt (s)
       END ).
```

The transmission of the actual-parameters is thus similar to the elaboration of identity-declarations (4.4.2.a); see also establishment (3.2.2.b) and ascription (4.8.2.a).}

{{Minor changes are required at other places in the Report.}}

{{The third bullet of 5.4.2.2 (semantics of formulas) is replaced by}}
• let R1 be {the routine which is} the yield of the parametrization {5.4.3.2.a2} of R with V1, ... , Vn;
• W is the yield of the calling {5.4.3.2.b} of R1 in E1;

{{5.4.4.2.case B, 10.3.4.1.2.c and 10.3.4.9.2 must be modified to show that the routines there created are composed, additionally, from a vacant locale {2.1.1.1.b}.}}

AB39.3.2 A Summary of the Replies to the ALGOL Bulletin questionnaire

B. A. Wichmann

Sixteen replies were received, twelve from implementors and four from interested users. For each of the fifteen features, they were asked (a) does your implementation already include this feature, (b) would the implementation be invalidated by the change, (c) do you approve of the proposed change, (d) if the proposed change were made official, is it probable that your implementation would be brought into line? The summary is as follows:

Item number	(a) already			(b) invalidate			(c) approve			(d) change		
1	9Y	5N	:	0Y	12N	:	16Y	0N	:	1Y	3N	:
2	7Y	7N	:	2Y	10N	:	15Y	1N	:	1Y	5N	:
3	6Y	8N	:	1Y	10N	:	14Y	2N	:	1Y	5N	:
4	8Y	5N	:	5Y	6N	:	13Y	2N	:	0Y	4N	:
5	1Y	13N	:	10Y	3N	:	9Y	7N	:	1Y	8N	:
6	10Y	4N	:	4Y	7N	:	11Y	3N	:	2Y	1N	:
7	12Y	3N	:	1Y	11N	:	16Y	0N	:	3Y	0N	:
8	12Y	3N	:	2Y	11N	:	15Y	1N	:	1Y	0N	:
9	13Y	2N	:	1Y	12N	:	16Y	0N	:	1Y	2N	:
10	2Y	13N	:	7Y	5N	:	5Y	11N	:	1Y	7N	:
11	5Y	8N	:	4Y	7N	:	9Y	5N	:	1Y	7N	:
12	14Y	1N	:	0Y	13N	:	15Y	1N	:	1Y	1N	:
13	2Y	11N	:	5Y	7N	:	14Y	2N	:	3Y	6N	:
14	2Y	10N	:	8Y	4N	:	6Y	8N	:	1Y	7N	:
15	2Y	11N	:	8Y	5N	:	9Y	6N	:	1Y	7N	:

Number of questionnaires 16

Hence there was unanimous approval for static own and comments including characters (items 1 and 7). There was unanimous approval for removing integer labels (item 9) even though it has been implemented. Similarly, substantial approval was given to a number of items which would invalidate only a small minority of systems. In this category are: only fixed bounds to own arrays (item 2), own variables initialised to zero or false (3), controlled variable to remain defined on exit (6), strings to consist of characters rather than ALGOL basic symbols (8), complete specification of formal parameters (12), environmental enquiries (maxreal, minreal, maxint and epsilon, item 13). This leaves six items upon which opinions were divided.

4. step expression evaluated once per loop.
   This is an issue upon which views have differed for many years. The formulation proposed has approval from all but two replies. It is relatively efficient and simple to define.

5. Controlled variable cannot be subscripted.
   The majority approved of this change in spite of the fact that only one implementor said he had the feature. A majority of the working group favours this change, and hence Modified ALGOL 60 excludes subscripted control variables.

10. <integer>↑<negative integer> undefined.
    We were perhaps a little too radical here but the formulation has the advantage of not requiring a change in the subsets. An alternative formulation similar to the one in the ECMA subset was suggested but WG2.1 decided to retain the proposal in the ALGOL Bulletin.

11. goto undefined switch designator undefined.
    Again a majority approved although a minority of compilers contained this feature (according to the replies). WG2.1 approved of this change in a vote taken at Breukelen (1974).

14. IFIP input/output.
    These do not seem to be much liked. However, many more approved of them than had them in their implementation. As a simple, rudimentary system rather than a complete system there does not seem to be any alternative. There appeared to be a misunderstanding that we were proposing this as a complete I/O system. We are not, but merely providing a basic system from the existing IFIP procedures.

15. Additional procedures outterminator, fault and stop.
    Again a majority approved in spite of their absence in current systems.

It would be unwise to read much more than this into the replies since some inconsistencies exist.

At the last WG2.1 meeting at Munich, it was decided that after various amendments had been made, the document should be published as a "Supplement to the ALGOL 60 Revised Report". This supplement proposes changes to the Revised Report resulting in a "Modified Report on the Algorithmic Language ALGOL 60". A last minute amendment includes the ability to concatenate

strings. This is to allow spaces to stand for themselves within a string as proposed in the Draft ISO Technical Report 1972.

We should like to thank those who replied to the questionnaire, namely: A.J. Amorison, P. Bacchus, Lars Blomberg, W.M. Gentleman, Sakari Hayrynen, D.J. Leigh, Anne Rogers, Andrew J. Skinner, J.F. Smith, J.R.W. Smith, P.D. Stephens, Grace H.J. Sturgess, G.A. Tebling, Martyn Thomas, Kenneth G. Walter and T.P.T. Williams. Additional correspondence was received from D.J. Cairns, C.A.R. Hoare, R.S. Scowen, D. Simpson, Garry J. Tee and J.W. van Wingen.

AB39.4.1

MODULAR PROGRAMMING IN ALGOL 68

I F Currie
18 6 75

1 Design aims

A. The modular compilation system should aid and encourage one to write well structured top-down programs. For example, the following "module" (not 68-R) could be regarded as a model for a large class of problems:

```
program = ( int in;
            input (in);
            int answer;
            process (in, answer);
            output (answer)
          );
```

Our compiling system should be able to compile this (or something very like it) without reference to what input, process and output actually do, or how they are constructed. We should be able to run program by applying any one of a set of actual modules (compiled later) into each of these formal positions. These actual modules should be able to "see" only those indicators presented to them in program, eg an actual module for output should know what answer is but not what in is, and certainly nothing about the dynamic construction of program.

B. To be forced to always observe a top-down mode of working would be quite intolerable for certain programming tasks. For example, the creation of libraries of procedures is naturally a bottom-up activity. Similarly the abstraction of data structures by providing a suite of procedures operating on concrete data objects follows bottom-up thinking. ALGOL 68 is not very good for this kind of abstraction since a change in the mode of the concrete data object will generally invalidate any previously compiled module using the abstraction. This rather weakens the reasons for abstracting in the first place. However, the module which defines the abstraction should be one which allows other modules (as yet unwritten) to use some set of modes, procedures or values declared within it. Note that the current 1900 68-R module system is entirely bottom-up.

C. The interfaces between modules (ie those indicators passed down at the formal positions in the top-down case and those passed up in the bottom-up case) should be concise and explicit. In other words, the fact that two modules are fitted together does not imply that one module knows all the indicators of the other, but only those which are explicitly mentioned. The purpose of both the top-down and bottom-up approach is to minimise the possibility of error by restricting information flow along well defined directional channels.

D. The only factor in deciding whether two independently compiled modules fit together is that both agree on their mutual interface. Clearly it must be impossible to fit together incompatable modules.

E. The recompilation of a module should only invalidate a previous compilation of another module if it significantly changes their mutual interface. The meaning of significant in this context will depend on the level at which the linkage of modules is performed. In our system where the linkage is done at a quasi-binary level, the only insignificant change to the interface is its extension by addition of other indicators leaving the original indicators unchanged in mode and name.

2 The evolution of the system

2.1 Top-down

It is fairly clear that the top-down requirement (as expressed by our model program) could be met by existing procedural structures. Thus the module program could be a procedure with three procedure parameters:

```
proc program = (proc (ref int) input, proc (int, ref int) process, proc (int) output):
                (etc)
```

To run program, we simply call it with the appropriate actuals of the correct mode. If these actual modules were themselves structured, then it would be necessary to partially parameterise them to get the modes straight. The modules in this system are simply pure routine-denotations and a module which can be run as an independent program is simply a proc void. The elegance of this system is somewhat offset by the following snags:

1 We cannot pass mode and operator indicants as parameters of procedures.

2 I feel that it is preferable to pass information between modules by name and mode rather than position and mode as in proc calls. This is particularly true where different groups of programmers are involved.

3 The routine denotations for modules tend to have rather cumbersome and opaque parameter packs. Further, partial parameterisation is a very heavy-handed way of doing what will generally be a quite simple linkage job.

In order to get over these snags and still preserve the elegance of the original procedural construction, two new unitary clause constructions have been introduced. The first replaces the formal proc parameter call (eg input (in) in program) and at the same time defines its interface; the other replaces the call of a procedure which is a module (eg program itself). Thus, the module program becomes (now in 68-R 2900)

```
program = (int in;
           formal input (in);
           int answer;
           formal process (in, answer);
           formal output (answer)
          )
```

A runnable program could then be constructed by supplying actuals to the formals of program:

```
runnable program 1 = (call program (input = (read(in)),
                                    process = (answer := 2 * in),
                                    output = (print(answer))))
```

The bodies substituted for input etc in this call of program would more usually be independently compiled thus:

```
input module = at input of program (read(in))
```

making another runnable program by compiling:

```
runnable program 2 = program (input = input module, ... etc)
```

I shall call input of program the context of input module. Note that the call

construction is just another unitary clause and the bodies directly supplied for substitution to its parameters as in runnable program 1 can access any indicator normally available to that unitary clause.

2.2 Bottom-up

The bottom-up module seems to be adequately covered by saying that it is an unbracketed list of declarations terminated by a keep list of indicators to define the interface, eg

```
lists =
mode element = union (int, list),
     cell = struct (element hd, tl),
     list = ref cell;
list nil = nil;
proc cons = (element a, b) list: cell := (a, b);
proc hd = (ref list a) element: hd of a;
proc tl = (ref list a) element: tl of a;
keep element, list, nil, cons, hd, tl
```

The indicators kept in lists may be used in any other module which makes the new declaration

```
access lists
```

The range of these indicators will be the same as that of any indicator declared at this point. This sounds simple, but unfortunately hides a multitude of pitfalls. These are mainly due to difficulties in deciding just when the declarations in such a module are elaborated. If we say that the declarations are elaborated at the access, then different modules using lists would have different declarations for each of its kepts; also, the scope of their values might be unduly restricted. On the other hand, if the declarations are to be elaborated at some global level it appears that an ordering must be applied to the elaboration of different modules accessed in a program, since presumably one such module can access another. Note that this ordering is not only due to one module directly accessing another but also, because of the possibility of side effects, of both accessing a third module.

On the 1900 68-R system this ordering was defined by the order of the original compilation of the modules; this ordering must be considered unsatisfactory as it was the root cause of most of our difficulties in the practical use of the system. In order to side-step this dilemma, I propose to allow only those declarations which can be elaborated at compile-time to form part of a bottom-up module. This is not quite as drastic as it sounds; the allowable declarations are declarations of modes without dynamic parts, procedure declarations, identity declarations identifying constants, and accesses to other modules. The module "lists" is a legal bottom-up module.

Another point that arises in the bottom-up mode of working is the definition of the bottom. Clearly the module "lists" can be accessed from anywhere, but one could also imagine a similar module which could only be accessed in some more limited context. For example we might wish to write a suite of procedures in the context input of program which could only be accessed in one or more of the actuals supplied for input of program. The only difference it would make in this instance is that the procedures in the suite could use the variable in, derived from program.

2.3 The System context

It would be highly desirable if we could implement all of the standard prelude and system library using the same ideas and software as for user-defined modules. In order to do this, all user modules are considered to be compiled in a global system context which declares the variables in the standard prelude. There are very few of these variables (standin and standout are the most important), and they form the interface which is the system context. As this context implicitly surrounds all user modules, all of the variables in the system context are automatically available to all modules; similarly indicators kept in bottom-up modules in the system library will be automatically available - the appropriate access declaration will be inserted where necessary. The system module which provides the global context will look something like:

```
system
(
 access open module; co introduces the mode charput, and procedures open and close;
                     the access is not strictly necessary
                     since open module is in the system library co
 charput standin, standout;
 open (standin, etc); open (standout, etc);
 formal user (standin, standout);
 close (standin); close (standout)
)
```

In order to run a user module, x say, the compiler/loader will generate a call system (user = x).

Summary

The syntax of a module is:

Module → moduleid1 = {Context} Body;

Context → at formalid1 of moduleid2;

The module given by moduleid2 must have already been compiled and must contain a formal call on formalid1 which will give a set of indicators to the body.

Body → Tdbody,

Bubody,

Completed module;

Tdbody → Closed clause;

A Tdbody may contain formal calls and its Closed clause is voided.

Bubody → Constdeclist Keep Indicatorlist;

A Bubody may not contain formal calls.

Constdeclist is a sequence of constant declarations (see 2.2) separated by semi-colons.

In addition the syntax of Unitary clause is extended by:

Unitary clause → call Completed module

Formalcall;

Completed module → moduleid3 {(Formal identification list)};

Formalidentification → formalid2 = Closed clause,

formalid3 = Completed module;

Moduleid3 must have already been compiled (either at the system context or at formalid1 of moduleid2), has a Tdbody, and all of its formals must be in the formal identification list. The indicators specified by formalid2 in moduleid3 are available in the closed clause. The outer module in the completed module identified with formalid3 must have been already compiled at formalid3 of moduleid3.

Formalcall → formal formalid4 (Indicator list);

formalid4 becomes a formal of moduleid1. When formalid4 is identified with an actual, the indicators in the Indicator list become available to that actual.

Also the syntax of Declaration is extended by:

Declaration → access moduleid4;

Moduleid4 must have already been compiled (at either the system context or at formalid1 of moduleid2) and have a Bubody. The indicators kept in that Bubody are available in the current range.

Indicators arising from the standard prelude or system library will be available to modules without extra context or access declarations.

Proposal for a Modules facility in ALGOL 68.

C.H.Lindsey.

CHL 75-05-12

This proposal arises from Schuman's paper "Toward Modular Programming in High-Level Languages" (AB37.4.1), together with discussions with Schuman, Boom, Bourne and Guy in Manchester during May 1974, with Dewar at Breukelen, at the Support Sub-committee in Boston in January 1974, and afterwards with Dewar in Chicago. The proposal is described mainly by way of examples. A formal specification is given afterwards.

1. Module declarations.

A module-declaration declares a module-indication and ascribes a module to it. A module consists of a module-text and an environ (cf. routines).

```
module a = (int i := 0; proc c = (int j) void: i +:= j);
```

2. Invocations.

An invocation identifies a module-text declared earlier (possibly in a different compilation).

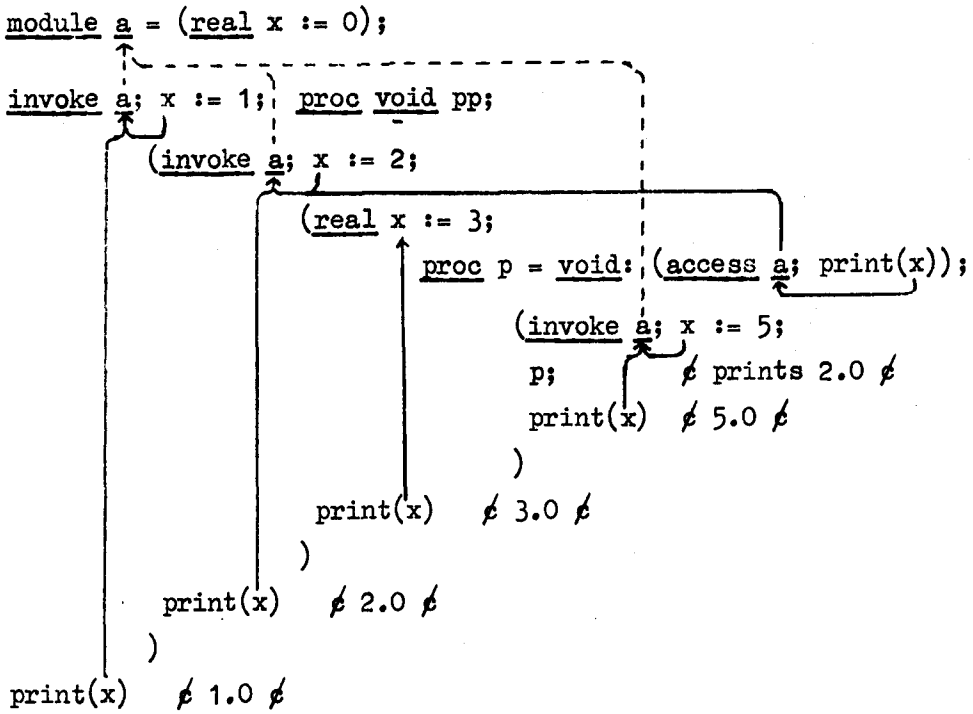
```
invoke a;
```

Basically, an invocation is a declaration that causes certain indicators to become visible in the current range and also, when it is elaborated, causes space to be reserved and ascriptions and initializations to be performed (just like any other declaration in the language). It follows that a given module may not be invoked at more than one place within a given reach, nor may it attempt to declare indicators already known within that reach.

3. Accessions.

An accession identifies an invocation made earlier (an invocation is thus both an applied occurrence and a defining occurrence of its module-indication). It causes the indicators declared earlier to become visible again (overriding any intermediate re-declarations of those indicators), but it does not involve any re-elaboration of their declarations.

The example below shows how a module-declaration is identified by invocations (dotted lines), and how invocations are identified by accessions and by other indicators (full lines).



Observe how the identification of "access a" determines the scope of the procedure "p", so that "pp := p" would have been illegal. Observe also that both module- and invocation-definitions of "a" occur in the outer reach. This is allowable because, even though these are both defining occurrences of "a", the 'QUALITY's so defined are not related to each other.

4. Hidden.

Within a module-text, declarations may be hidden (labels are hidden automatically). Such declarations are elaborated and are visible from within the module-text, but they are not rendered visible outside the invocation. If a module-text declares a variety of procedures, these may communicate amongst themselves by means of hidden global variables, with no possibility of the user interfering with those variables himself. This is comparable to, but more powerful than, the own facility of ALGOL 60.

The module r below, when invoked, makes available a procedure random, but not the underlying variable which remembers the current position in the pseudo-random sequence.

```
module r = (hidden real lastrandom := 0.5;
            proc random = real: ( ... ; lastrandom := ... ));
invoke r;
print(random)  ¢ but not print(lastrandom) ¢
```

Lastrandom is not now visible, but its scope is clearly that required.

Frequently, in order to use the hidden facility, one has to declare a module which is then only invoked upon one occasion. This can be avoided by immediate invocation of a module-text.

```
invoke ( hidden real lastrandom := 0.5;
         proc random = real: ( ... ; lastrandom := ... ) ) ;
print(random)  ¢ but not print(lastrandom) ¢
```

This also makes it possible to render a whole group of declarations in a module invisible without having to write the word "hidden" in front of each one.

```
module a = (hidden invoke (int private1;
                           proc private2 = void: private1 := public2);
                    invoke (int public1;
                           proc public2 = void: (private2; print(private1))));
invoke a;
public1 := 2; public2
```

This feature would have been useful when writing the standard-prelude of the Revised Report.

5. Provocations.

Partial compilation of programs is needed for two purposes: for assembling large particular-programs piecemeal, and for constructing libraries for use by many particular-programs. The latter problem raises the particular difficulty of where to invoke a module that may be used at several places in a particular-program, perhaps including invocations of other library modules which use it at second hand. It may be important that it be invoked only once, so as to get only one copy of any global variables that it may create. Contrariwise, it must be invoked afresh in each particular-program - otherwise all particular-programs will share the same globals. Such library modules should therefore be conceived as having been declared in the particular-prelude, rather than in the library-prelude.

Suppose that the library modules 'vibrations' and 'stresses' both make use of the library module 'matrices'. However, the user of these modules is presumed to be unaware of the fact and will not therefore invoke 'matrices' himself (unless he needs it for some other purpose of his own). It is required that only one invocation of 'matrices' should be brought into existence, whether the user invokes 'vibrations', or 'stresses', or his own use of 'matrices', of any combination thereof. The particular-prelude therefore contains:

```
module matrices = ( ... );                                  provoke matrices;
module vibrations = (hidden access matrices; ... ); provoke vibrations;
module stresses   = (hidden access matrices; ... ); provoke stresses;
```

Here, "provoke matrices" serves as a defining occurrence for "access matrices" to identify, but its elaboration will not be triggered unless such an "access matrices" actually occurs (moreover, if the "access matrices" is itself a constituent of a module-definition, there must also occur an invoke or a triggered provoke to trigger that).

So, we now have:

```
begin ¢ of the particular-program ¢
access vibrations;
access matrices ¢ because the user wanted to use it for himself ¢ ;
...
end
```

whereupon the "provoke matrices" and the "provoke vibrations" (but not the "provoke stresses") in the particular-prelude are triggered (but the body of matrices is elaborated only once).

The normal convention should be for libraries to declare modules and to provoke them. Users should then use access rather than invoke, unless they specifically want a private copy of the module.

A provocation may also be used within a particular-program (here, it would be most unlikely not to be triggered) when it is required that declarations should be elaborated, but that their indicators should not be made visible until some inner range.

Note that an access always identifies an invoke or a provoke, and that an invoke or a provoke identifies a module.

6. Partial compilation.

Even though a particular-program is made up of several separately compiled parts, there must still exist a conceptual complete particular-program made of some concatenation of its constituent texts (parhaps with a few enclosing begins and ends provided by the system). This conceptual complete particular-program is the one whose meaning will then be defined by the Report. The manner in which pre-compiled modules are filed away and retrieved by the operating system is not defined in this proposal, but should be controlled by pragmats.

```
begin
module a pr in album A pr = ( ... );                    ] 1st compilation
module b pr in album B pr = ( ... );                    ] 2nd compilation
    ⋮                    ⋮                                  ⋮
        begin                                           ]
        invoke a pr from album A pr, b pr from album B pr;  ] final compilation
        ...                                             ]
        end                                             ]
end
```

Modules may make use of declarations in other modules provided, presumably, that they do not ask for information in modules not yet compiled.

```
begin
module a = begin                                  ┐
           proc void pp;                          │
           real y;                                │
           (invoke b; skip);                      │ 1st compilation
           pp                                     │
           end;                                   ┘
module b = begin                                  ┐
           access a;                              │
           proc p = void: print(y);               │ 2nd compilation
           pp := p                                │
           end;                                   ┘
invoke a; skip                                    ┐ final compilation
end                                               ┘
```

In the second compilation, access a identifies (with some help from the loader) the invoke a right at the end, but it relies on the previous compilation of module a for the declarations of "pp" and "y". Although the first compilation invokes b (which is not yet compiled), the declarations of b are not needed. Provided the compiler is able to recognise the special case "(invoke b; skip)", no problems need arise.

Formulation of module proposal.

{{Module declarations and invocations are new kinds of declarations. New kinds of entry in the nest are therefore needed.}}
1.2.3.
E) DEC :: ... ; MODULE TAB ; SUBSIDIARY TAB.
L) MODULE :: module DECSETY TALLY.
M) SUBSIDIARY :: subsidiary DECSETY TALLY.
4.8.1.
F) QUALITY :: ... ; MODULE ; SUBSIDIARY.
{{'MODULE TAB's will be introduced into the nest by module-declarations. 'SUBSIDIARY TAB's will be introduced by INVOCATIONs.}}

{{New kinds of indicator are needed to identify these new properties.}}
4.8.1.
A) INDICATOR :: ... ; module indication.

{{Modules are ascribed to module-indications by means of module-declarations.}}

4.9. Module declarations

4.9.1. Syntax

a) NEST1 module declaration of DECS{41a} :
module{94d} token, NEST1 module joined definition of DECS{41b}.
b) NEST1 module definition of module DECSETY TALLY TAB{41b} :
module DECSETY TALLY NEST1 defining module indication
with TAB{48a},
where <TAB> is <bold TAG>,
is defined as{94d} token,
NEST1 TALLY module text exhibiting DECSETY
defining new DECSETY PROPSETY{c}.
c) NEST1 TALLETY i module text exhibiting DECSETY
defining new DECSETY PROPSETY{b,4Ab} :
NEST1 new DECSETY PROPSETY TALLETY module series
with DECSETY without PROPSETY{d} STYLE pack.

d) NEST2 TALLETY module series with DECSETY without PROPSETY{c} :
strong void NEST2 unit{32d}, go on{94f} token,
NEST2 TALLETY module series with DECSETY without PROPSETY{d} ;
where <DECSETY> is <DECSETY1 DECSETY2>,
NEST2 TALLETY1 declaration of DECSETY1{41a}, go on{94f} token,
NEST2 TALLETY2 module series
with DECSETY2 without PROPSETY{d},
where TALLETY is the greater of TALLETY1 and TALLETY2{e} ;
where <PROPSETY> is <DECSETY1 DECSETY2 LABSETY2>,
hidden{94d} token,
NEST2 declaration of DECSETY1{41a}, go on{94f} token,
NEST2 TALLETY module series
with DECSETY without DECSETY2 LABSETY2{d} ;
where <DECSETY cum PROPSETY> is <EMPTY cum LAB LABSETY>,
NEST2 label definition of LAB{32c},
NEST2 module series with EMPTY without LABSETY{d} ;
where <DECSETY cum PROPSETY> is <EMPTY cum LAB LABSETY>,
completion{94f} token, NEST2 label definition of LAB{32c},
NEST2 module series with EMPTY without LABSETY{d} ;
where <DECSETY cum PROPSETY> is <EMPTY cum EMPTY>,
EMPTY.

e) WHETHER TALLETY is the greater of TALLETY1 and TALLETY2{d,41a,b} :
where <TALLETY> contains <TALLETY1>,
WHETHER <TALLETY> is <TALLETY2> ;
where <TALLETY> contains <TALLETY2>,
WHETHER <TALLETY> is <TALLETY1>.

f)* module text defining LAYER :
NEST TALLY module text exhibiting DECSETY defining LAYER{c}.

{The use of 'TALLY' excludes circular chains of module-definitions such as MODULE A = (INVOKE B; SKIP; HIDDEN REAL x), B = (INVOKE A).}

{Examples:
a) MODULE A = (REAL a, b; a := b := 0),
B = (REAL a, b; HIDDEN REAL x, y)
b) B = (REAL a, b; HIDDEN REAL x, y)
c) (REAL a, b; HIDDEN REAL x, y)
d) REAL a, b; HIDDEN REAL x, y }

{{In example c, 'DECSETY' corresponds to the declaration REAL a, b and 'PROPSETY' to the hidden declaration REAL x, y. Both these properties will be made visible as soon as this module text is invoked, except that 'PROPSETY' will only be visible from inside it.}}

4.9.2. Semantics

a) A module-declaration D is elaborated as follows:
. the constituent module-texts of D are elaborated collaterally;
For each constituent module-definition D1 of D,
. the yield of the module-text of D1 is ascribed {4.8.2.a} to the defining-module-indication of D1.

b) The yield of a module-text T, in an environ E, is the scene composed of
(i) T, and
(ii) the environ necessary for {7.2.2.c} T in E.

c) A module-series C is elaborated as follows:
If C has no direct descendent {i.e., it is EMPTY}, or if C contains a direct descendent completion-token,
then the elaboration of C is completed;
otherwise,
. the declaration or the unit, if any, of C is elaborated;
. the series of C is elaborated.

{{It is now possible for a single COMMON-definition to create more than one 'DEC' in the nest, or even no 'DEC's at all. Moreover, 'TALLY's must be passed back from certain COMMON-definitions to the module-series.}}

4.1.1.
A) COMMON :: ... ; module ; INVOCATION.

a) NEST TALLETY declaration of DECSETY{a,32b,49d} :
NEST TALLETY COMMON declaration of DECSETY{42a,43a,44a,e,45a, 49a,4Aa,-} ;
where <DECSETY> is <DECSETY1 DECSETY2>,
NEST TALLETY1 COMMON declaration of DECSETY1 {42a,43a,44a,e,45a,49a,4Aa,-},
and also{94f} token,
NEST TALLETY2 declaration of DECSETY2{a},
where TALLETY is the greater of TALLETY1 and TALLETY2{49e}.
b) NEST TALLETY COMMON joined definition of PROPSETY {b,42a,43a,44a,e,45a,46e,49a,4Aa,541e} :
NEST TALLETY COMMON definition of PROPSETY {42b,43b,44c,f,45c,46f,49b,4Ab,541f,-} ;
where <PROPSETY> is <PROPSETY1 PROPSETY2>,
NEST TALLETY1 COMMON joined definition of PROPSETY1{b},
and also{94f} token,
NEST TALLETY2 COMMON definition of PROPSETY2 {42b,43b,44c,f,45c,46f,49b,4Ab,541f,-},
where TALLETY is the greater of TALLETY1 and TALLETY2{49e}.

{{Rule 4.1.1.c is no longer needed.}}

{{This requires some consequent changes in the syntax of series.}}

3.2.1.
b) SOID NEST series with PROPSETY{a,b,34c} :
... ;
where <PROPSETY> is <DECSETY1 DECSETY2 LABSETY2>,
NEST TALLETY declaration of DECSETY1{41a}, go on{94f} token,
SOID NEST series with DECSETY2 LABSETY2{b} ;
... .

{{Modules may be invoked by invocation-declarations. Modules already invoked in this way may also have their indicators rendered visible again by means of accession-declarations.}}

4.10. Invocation declarations

4.10.1. Syntax

A) INVOCATION :: invocation ; provocation ; accession.
B) MODIARY :: module DECSETY ; subsidiary DECSETY.

a) NEST2 TALLY INVOCATION declaration of DECSETY{41a} :
INVOCATION{94d} token,
NEST2 TALLY INVOCATION joined definition of DECSETY{41b}.
b) NEST2 TALLY INVOCATION definition of DECSETY{41b} :
where <NEST2> is <NEST1 new PROPSETY1 DECSETY PROPSETY2>
and DECSETY independent PROPSETY1 PROPSETY2{71a,b,c,q,r},
where INVOCATION of MODIARY TALLY TAB defines
DECSETY{c,d,e,-},
MODIARY TALLY NEST2 applied module indication with TAB{48b} ;
where <NEST2> is <NEST1 new PROPSETY1 DECSETY PROPSETY2>
and DECSETY independent PROPSETY1 PROPSETY2{71a,b,c,q,r},
where <INVOCATION> is <invocation>,
NEST2 TALLY module text exhibiting DECSETY
defining new DECSETY PROPSETY{49c}.
c) WHETHER invocation of module DECSETY TALLY TAB defines
DECSETY subsidiary DECSETY TALLY TAB{b} : WHETHER true.
d) WHETHER provocation of module DECSETY TALLY TAB defines
subsidiary DECSETY TALLY TAB{b} : WHETHER true.
e) WHETHER accession of subsidiary DECSETY TALLY TAB defines
DECSETY{b} : WHETHER true.

{Examples:
a) INVOKE B, (REAL a, b; a := b := 0)
b) B . (REAL a, b; a := b := 0) }

{Note that a provocation places a 'subsidiary DECSETY TALLY TAB' in the nest. This may subsequently be identified by an accession which then makes those 'DECSETY' available. An invocation makes both the 'subsidiary DECSETY TALLY TAB' and the 'DECSETY' available together, so that INVOKE A is identical in its effect to PROVOKE A; ACCESS A.}

7.1.1.
o) WHETHER MODULE related SUBSIDIARY{d} : WHETHER false.
p) WHETHER SUBSIDIARY related MODULE{d} : WHETHER false.

{{Observe that 'MODULE' is not related to 'SUBSIDIARY', so that MODULE A = (SKIP) and INVOKE A may both occur in the same reach. Contrariwise, this means that in
(MODULE A = (SKIP); INVOKE A; (MODULE A = (SKIP); ACCESS A))
the ACCESS A in the inner reach identifies the INVOKE A in the outer one.}}

4.10.2. semantics

a) The elaboration of an INVOCATION-declaration consists of the collateral elaboration of its constituent INVOCATION-definitions.

b) An INVOCATION-definition-of-DECSETY I, in an environ E, is elaborated as follows:
. let S be the scene yielded, in E, by the applied-module-indication or module-text of I in E;
Case A: I is an invocation-definition or a "triggerred" {c} provocation-definition:
. let E1 be the environ established beside E {3.2.2.b}, around the environ of S, according to the module-text T of S;
. the module-series of T is elaborated in E1 {4.9.2.c};
If the direct descendent of I is an applied-module-indication, then
. let 'DECSETY' be 'DECSETY1 SUBSIDIARY TAB';
. the scene composed of T and E1 is ascribed in E to some SUBSIDIARY-defining-module-indication-with-TAB;
Case B: I is an accession-definition:
. let E1 be the environ of S;
. let 'DECSETY1' be 'DECSETY';
Other cases {I is an untriggerred provocation-definition}:
. the elaboration of I {involves no further action and} is completed;
If 'INVOCATION' is 'invocation' or 'accession',
then
For each value or scene which has been {previously in the case of 'accession'} ascribed in E1 to a QUALITY-defining-indicator-with-TAX J,
If 'DECSETY1' envelops 'QUALITY TAX',
then that value or scene is {re-}ascribed in E to J.

c) A provocation-definition I is "triggerred" if there exists some triggerred accession-definition whose applied-module-indication identifies {7.2.2.b} the applied-module-indication of I.
An accession-definition is "triggerred" if it is not a constituent of an untriggerred module-definition.
A module-definition M is "triggerred" if there exists some invocation-definition {whether triggerred or not} or some triggerred provocation-definition whose applied-module-indication identifies {7.2.2.b} the defining-module-indication of M.

{The elaboration of an untriggerred provocation-definition involves no action. For example, in the elaboration of
MODULE A = (REAL x := random); PROVOKE A;
MODULE B = (HIDDEN ACCESS A; REAL y := x * random); PROVOKE B;
MODULE C = (HIDDEN ACCESS B; REAL z := y * random); PROVOKE C;
BEGIN ACCESS B; print (y) END
random is called only twice.}

{{Establishment "beside" an environ (as opposed to "upon" it) requires a change to 3.2.2.b. The first bullet of that rule becomes:}}
. upon or beside an environ E1, possibly not specified, {which determines its scope,}
{{The two bullets commencing "if E1 is not specified ..." become:}}
. if E1 is not specified, then let E1 be E2 and let "upon E1" be assumed;
. E is newer in scope than E1 (is the same in scope as E1) if the establishment is upon E1 (is beside E1) and is composed of E2 and a new locale corresponding to 'PROPSETY', if C is present, and to 'EMPTY' otherwise;

{{Various new symbols have been invented:}}
9.4.1.d

module symbol{49a}	MODULE
hidden symbol{49d}	HIDDEN
invocation symbol{4Aa}	INVOKE
provocation symbol{4Aa}	PROVOKE
accession symbol{4Aa}	ACCESS

{{Minor changes are required at other places in the Report.}}

{{3.2.2.b.Case A. The line specifying the construct of the scene ascribed for a label-definition becomes:}}
(i) the series or module-series of which L is a direct descendent, and

{{A jump might be a constituent unit of a module-series rather than of a series. The first bullet of 5.4.4.2 becomes:}}
. let the scene yielded in E by the label-identifier of J be composed of a series (a module-series) S2 and an environ E1;
{{The first bullet of 5.4.4.2.Case A becomes:}}
. let S1 be the series (module-series) of the smallest {1.1.3.2.g} serial-clause (module-text) containing S2;

{{Extra predicates.}}
7.1.1.
q) WHETHER PROPS PROP independent PROPSETY{4Ab} :
WHETHER PROPS independent PROPSETY{a,b,c,q,r}
and PROP independent PROPSETY{a,b,c}.
r) WHETHER EMPTY independent PROPSETY{4Ab} : WHETHER true.

{{The proper identification of indicators declared via invocations is ensured as follows:}}
7.2.2.
b) The defining NEST-range {a} of each QUALITY-applied-indicator-with-TAX I1 contains {of necessity} either a QUALITY-NEST-LAYER-defining-indicator-with-TAX I2, or else an applied-module-indication I3 directly descended from a NEST-LAYER-INVOCATION-definition-of-DECSETY1-QUALITY-TAX-DECSETY2. I1 is then said to "identify" that I2 or I3.
{{This is sufficient to ensure, in conjunction with 7.2.2.c, the proper scope for routines containing accessions.}}

{{1.1.4.2.c. The list of elidible hypernotions must include:}}
... "without DECSETY LABSETY" . ...

{{An extra predicate:}}
1.3.1.
o) WHETHER <EMPTY> contains <EMPTY> : WHETHER true.

{{Revised pragmatic remark concerning scopes:}}
2.1.1.3.
b) Each environ has one specific "scope". {The scope of each environ is never "older" (2.1.2.f) than that of the environ from which it is composed (2.1.1.1.c).}

{{A module-text must be an establishing-clause.}}
3.2.1.
i)* establishing clause : ... ;
NEST TALLY module text exhibiting DECSETY defining LAYER{49d}.

{{Library declarations (especially those which have side effects) should be in the particular-prelude rather than in the library-prelude. In 10.5.1, extend the first sentence by:}}
and forms which, because of their side effects, are inappropriate for inclusion in the library-prelude {10.1.2.c}.

AB39.4.3 A Note on Integral Division

by L.G.L.T. Meertens, Mathematisch Centrum, Amsterdam.

Editor's note This paper is taken from a letter by Lambert Meertens in reply to someone who had pointed out that the mod operator in ALGOL 68 does not provide the same result as the remainder implied by the ÷ operator. This is a sad tale, involving a lot of history going back to the design of ALGOL 60 and the fact that most American computers worked in "sign and modulus" notation at that time. The matter has been raised before (see AB28.3.2, L04 and L05, half of which was accepted) but it has always been difficult to excite any concern over it. The conclusion of the present paper is that it is essentially the ÷ operator which is wrong. Future language designers please note.

My distrust for arguments based on "natural choice for widely spread computers" almost parallels my dislike for inconsistency.

i) The origin of integral division lies in the following question:
How many times b may be taken from a?
or, in a more mathematical expression
max {q | q ≥ 0 ∧ q × b may be taken from a}
or max {q | q ≥ 0 ∧ q × b ≤ a}.

If we abbreviate this to a quot b (from Latin quotiens = how many times), we might define

```
op quot = (int a, b) int:
if a < 0 ∧ b > 0 then undefined
elif b ≤ 0 then undefined
else int q := 0 ;
     while (q + 1) × b ≤ a do q +:= 1 od ;
     q
fi .
```

Note that the reasons behind the undefinedness for a < 0 ∧ b > 0 and for b ≤ 0 are of a different nature: in the first case no natural q satisfies q × b ≤ a, in the second case no such maximal q exists.

Several ways exist to relax the undefinedness. A "natural" way would be to express the original question algorithmically thus:

```
int q := 0 ;
while b may be taken from a
do (take b from a, q +:= 1) od
```

which would have a quot b equal to zero for a ≤ b.

Another direction is indicated by algebraic considerations, viz, by the wish to extend the validity of a = q × b → a quot b = q from natural a and positive b to arbitrary integral a and non-zero b.
One way to obtain the desired result is to define

a ÷ b = sign a × sign b × (abs a quot abs b) ,

chosen in ALGOL 60/68 and the hardware of many a computer, but this is certainly not the only way. Arguments for this choice in ALGOL 68 were the compatibility with ALGOL 60 and the Bauer-Samelson criterion (since the "normal" question is that for which a is natural and b positive). It would, however, have been possible, and, I think now, have been desirable, to define the operation in such a way that

(a + n × b) ÷ b = a ÷ b + n

would have been valid for arbitrary a and n and non-zero b. For example, in the binary search algorithm, an assignation like mid := (left + right) ÷ 2 will occur, and it is clearly desirable that this is not sensitive to a simultaneous shift in the bounds. At present, if we define

```
op mid = (ref [ ] real x1) ref real:
          x1 [(lwb x1 + upb x1) ÷ 2 ],
```

then

```
[-1 : 0] real x1 ; mid x1 :=: mid x1 [@1 ]
```

yields false!

ii) The origin of the modulo-operation lies in algebra:
Given a positive b, the integral numbers ℤ may be split into b residue classes, denoted 0, 1, . . ., b - 1, where m = {n ∈ ℤ | n ≡ m (mod b)} .
We now want an operator mod such that a ∈ m ↔ a mod b = m.

Again, this may in some way be extended to arbitrary non-zero b, e.g., by using $\mathbb{Z}_b = \mathbb{Z}_{-b}$, since $\mathbb{Z}_b$ is the quotient group ℤ /{b}, where the ideal {b} = bℤ = (-b)ℤ = {-b}. Note that a is already arbitrary integral.

iii) The inconsistency.

Define a rem b = a - a quot b × b. ALGOL 60 programmers will have felt a need for such an operation. As soon as one is doing multi-length integral arithmetic, base conversion, etc., the remainder is as important as the quotient. In view of the "size" of arithmetic values and the conversion routines in ALGOL 68, the need will have disappeared largely. Moreover, unless a and b have opposite and different signs, the programmer may use

a rem b = a mod b.

The "inconsistency" is now that this does not hold for all a and b.

Possible remedies:

R_i: Redefine ÷. This is a clean solution, which has been adopted in ALEPH.

R_{ii}: Strike mod. But whom do we serve by this? Expressing mod by means of ÷ is cumbersome and bug-prone (since the programmer is likely to overlook the possibility of negative a), and is probably dealt with more efficiently in code.

R_{iii}: Add rem. This is also a clean solution. However, there seems to be little need for it, and whatever need is left will quite likely be concerned with natural a and positive b (which is catered for by mod), and, if not, the user will as likely want a quot b to yield zero for a ≤ b, and therefore, a rem b to yield a, as any other result.

It seems too late for any of these, but it will be clear that I should favour R_i. For future ALGOL 68-ish languages, I should like to see something like op (÷, ÷×) = (int a, b) struct (int, int) : c (quotient, remainder) c.

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 as ERRATA-3 in ALGOL Bulletin 38, and are to be applied after them. They bring the text of TR74-3 into line with the definitive text of the Revised Report, as published in Acta Informatica Vol. 5, pts 1, 2 & 3.

CATEGORY A (significant errors)

p121 10.2.3.0.a+5 # AND => & = 3, AND #

a+15 # I => +x = 9, +* = 9, I #

p122 10.2.3.2.b # AND => &, AND #

p123 10.2.3.3.u # I => +x, +*, I #

p124 10.2.3.4.s # I => +x, +*, I #

10.2.3.5.e,f # I => +x, +*, I #

p126 10.2.3.8.d # AND => &, AND #

p150 10.3.1.6.k+2:k+14 {supersedes ERRATA-2}

```
# ELSE BOOL reading ??? FI =>
ELSE REF REF POS cpos = cpos OF f;
     WHILE c OF cpos ≠ c
     DO
       IF c < 1 ∨ c > c OF book bounds (f) + 1
       THEN undefined
       ELIF c > c OF cpos   THEN space (f)
       ELSE backspace (f)
       FI
     OD #
```

p151 10.3.2.1.c+8

```
# ␣LONG REAL xx := y; => #
# length := 0 => length := (after = 0 | 1 | 0) #
```

c+9

```
# xx ≥ =>
y + ␣LONG .5 x ␣LONG .1 ↑ after ≥ #
```

p216 12.4."+=:"

```
# r, t => r, t
  +x 10.2.3.0.a, 10.2.3.3.u, 10.2.3.4.s,
     10.2.3.5.e, f
  +* 10.2.3.0.a, 10.2.3.3.u, 10.2.3.4.s,
     10.2.3.5.e, f
  & 10.2.3.0.a, 10.2.3.2.b, 10.2.3.8.d #
```

CATEGORY B (clarifications)

p39 2.1.4.2.-2:-1

```
# constituent reference-to-real-assignation
     (5.2.1.1.a) =>
  constituent reference-to-real-serial-clause
     (3.2.1.a) #
```

p45 3.0.1.f+3,+4 # NEST => NEST STYLE #

p46 3.2.+4 # sequence => possibly empty sequence #

3.2.+5 # which => which, if any, #

p63 4.6.1.-3 # the flexibility => flexibility #

p64 4.6.1.examples.i
1 : n, 1 : m => 1 : m, 1 : n

p65 4.6.2.a+4 # developed => "developed" #

p68 5.+7 # the␣end => the end # {twice}

p76 5.3.2.2.b.Case C+5
which => and # {twice}

p83 6.1.+10 # NEST => 'NEST' #

p88 6.7.1.-2 # call => calling #

p89 7.+5 # boolean => 'boolean' #

p90 7.1.1.m # k,l,n => k,l,n,47f #

p91 7.2.-11 # 'real letter x' =>
'reference to real letter x' #

p99 8.1.0.1.a+1 # SIZE{94d} symbol => SIZE symbol{94d} #

p101 8.1.2.2.b.Case A+2
␣V␣ => ␣V␣

p103 8.2.2.a+7 # in ␣D␣ => contained in ␣D␣ #

p104 8.3.+4 # all␣is␣well => all is well #

p107 9.2.1.d+6 # other PRAGMENT symbol => PRAGMENT symbol #

p114 9.4.2.2.c+2 # a typographical display feature =>
typographical display features #

9.4.2.2.c+7 # letter-l-letter-o => bold-letter-l-letter-o #

p119 10.2.1.j+3,m+3 # in ␣l␣, =>
of which '␣l␣' is composed, #

p121 10.2.3.0.a+7,+8 # < = 5 ??? GT = 5, =>
< = 5, LT = 5, ≤ = 5, <= = 5, LE = 5,
≥ = 5, >= = 5, GE = 5, > = 5, GT = 5, #

10.2.3.1.e+3 # produced => produced (nor any unintended
particular-program be produced) #

p135 10.3.1.3.bb+1,+3,+5,+7,+9
which yields => , which is

p137 10.3.1.3.cc."on char error"-3
"page␣number␣" => "page number "

p138 10.3.1.3.ff+1 # calling => ␣calls␣ of #

10.3.1.3.hh+5 # n; => n := 0; #

p140 10.3.1.4.aa+1 # an explicit call => explicitly calling #

p142 10.3.1.4.d+16:17
the book ??? putting =>
opening is inhibited by other users #

```
p147 10.3.1.6.b+2     # current pos(f) => cpos OF f #

p151 10.3.2.1.c+6     # y := => y = #

p157 10.3.3.1.a."l bits lb"+2
                      # ␣f␣ OF lb => ␣l f␣ OF lb #

p161 10.3.3.2.a."REF L BITS"+2
                      # ␣f␣ => ␣l f␣ #

p165 10.3.4.1.1.Examples.a,.b,.d,.e
                      # "table␣of" => "table of" #

p167 10.3.4.1.1.gg+1 # 10.2.5.a => 10.3.5.b #

     10.3.4.1.1.hh+2 # cc => dd #

     ii+2             # dd => ee #

p173 10.3.4.3.1.bb+22      # number to => number of digits to #

p175 10.3.4.6.1.Example.a
                      # "table␣of" => "table of" #

p176 10.3.4.7.1.bb+8 # . => ;
                       ● if ␣l● cannot be represented by such a
                       string, the conversion is unsuccessful. #

p182 10.3.5.d+7       # ELSE => ELIF UPB ins > 1
                        THEN #

     10.3.5.g-4       # ELSE => ELIF UPB frames > 1
                        THEN #

p185 10.3.5.1.a."edit L int"+2
                      # s := => STRING t = #

     +3               # LOC INT, s => LOC INT, t #

     +6               # ELSE => ELSE t PLUSTO s; #

     +7               # UPB s => UPB t #

     "edit L real"+2 # sign2 := FALSE, => STRING point := ""; #

     +3               # STRING t; => #

     +5               # )) => ); point := ".") #

     +6               # (? "e" ??? )); => #

     +7               # e * 0 => ? "e" #

     +8:+9            # (t := ??? PLUSTO s); => edit int (exp); # {sic}

     +12              # (t := => STRING t = #
                      # )) => ); #
     +13              # PLUSTO s; => #

     +14              # LOC INT, s => LOC INT, t #

     +17              # ELSE =>
                        ELSE t [ : b] + point + t [b + 2 : ] PLUSTO s; #
```

```
p187 10.3.5.1.a."GPATTERN"-3
                    # sinsert := LOC [1 : 0] SINSERT; =>
                      FOR i TO UPB sinsert
                      DO sinsert [i] := (0, "") OD; #

p188 10.3.5.1.b+10  # true => TRUE #

     'marker="."'+1 # (sj + "." | j -:= 1); => #

p189 10.3.5.2.a."¢ boolean ¢"+2
                    # STRING (flip) => flip #

     10.3.5.2.a."¢ complex ¢"+6
                    # b1 ∨ b2 => b1 ∧ b2 #

p191 10.3.5.2.a."FPATTERN"+2
                    # sinsert := LOC [1 : 0] SINSERT; =>
                      FOR i TO UPB sinsert
                      DO sinsert [i] := (0, "") OD; #

     "FPATTERN"+4   # END => END , #

p199 11.5.+3        # suitable => suitably #

p202,203
     11.11."putf"+1,+3,+5,+7 (at the end of each line)
                    # ␣ => "x #

     "putf"+2,+4,+6,+8 (at the start of each line)
                    #  => " #
                    {in other editions of the Report, this treatment
                    of line splits in formats may occur at different
                    positions}

     "putf"+1:+9    # ␣ =>   # {throughout}

p216 12.3."balances" # g => g
                      'begins with' 1.3.1.h, i, j #

     "coincides with"
                    # l => l
                      'contains' 1.3.1.m, n #

     "false"        # a, b => b #

     "subset of"    # n => n
                      'true' 1.3.1.a #

p220 "print","printf","read","read bin","readf"
                    # 10.5.2. => 10.5.1. #

     "standconv"    # 2.d => 2.d
                      stop 10.5.2.a #

     "write","write bin","writef"
                    # 10.5.2. => 10.5.1. #
```

CATEGORY C (minor misprints)

p4	0.2.3.+1	# ◆own• => ▫OWN• #
p18	1.1.3.4.e+10	# (b) may => (c) may #
p24	1.1.5.b.(iii)+2	# provided ones; => ones provided in ▪D•, #
p25	1.3.-1	# Looking => Looking #
p27	1.3.3.-5	# •'unless => ◆'unless #
p33	2.1.3.1.f-5	# to a => a #
p38	2.1.4.+2	# action => action. #
p46	3.1.1.a+1	# 32a => •32a◆ #
	3.2.1.a+3	# Here => •Here #
	b+1	# 94f => •94f◆ #
p48	3.2.2.b.Case A	# a an => an #
p52	3.4.1.-2	# except => (except #
p54	3.4.2.-11	# 1664 => 1664, #
p57	3.5.2.step 3+2	# ▪while => ◆while #
p61	4.4.2.-2	# • => x #
	4.4.2.b+6	# 'MODE'• => ◆'MODE'• #
p65	4.6.2.b.Case B+7	# bound◆of that => bound• of that #
p71	5.2.3.1.-1	# ▫sample => ◆sample #
p84	6.1.1.Examples.e+1	# cross references => cross-references #
p88	6.7.+5	# • => x #
p90	"ambiguities"+1	# 9? ¢•=> 9? ¢• , #
p91	"In such cases"-5	# SKIP• => SKIP• , #
	-1	# 2? ¢• => 2? ¢• . #
	7.2.-1	# end• } => END• .} #
p93	7.2.2.c+3,+12	# , => ; #

p102 8.1.5.+1 # •void => +void #

p107 9.3.c-2 # reference language => reference-language #

p113 9.4.2.1.G # H => •H+ #
I => •I+

9.4.2.2.b+5 # being, => , being, #

p128 10.2.3.9.e # ■L BYTES => ■L □BYTES #

p125 10.2.3.7.0.+1 # short => shorten # {twice}

p138 10.3.1.3.ff+2 # call => +call• #

p140 10.3.1.4.bb-1 # implementation dependent =>
implementation-dependent #

p147 10.3.1.6.dd # call => +call• #

p156 10.3.3.1.ee # , then => •, then #

p168 10.3.4.1.1.kk-6 # suppresssed => suppressed #

p159 10.3.3.2.cc+4 # "□i•" or "□i•" =>
□"i"• or □"i"• #

p171 10.3.4.1.2.d+3 # i = 1 => ■i = 1 #

p177 10.3.4.8.1.b+4 # token{94f} => {94f} token #

p185 10.3.5.1.a."edit L real"+14
errrorchar => errorchar

p196 10.5.1.+7 # TAX+ contained => TAX• contained #

p197 10.5.2.+5 # calls => +calls• #

p214 "unsuppressible-suppression"
10.3.4.1.1.l => •10.3.4.1.1.l

p217 12.4."*" # 10.2.3.10.l, m, n, o, => #

"*"+1 # s => s, 10.2.3.10.l, m, n, o #

"**" # t, t => t #

"%+" # %+ 10.2.3.11.k => #

"%+:=" # %+:= 10.2.3.0.a => #

"%x" # n => n
%x:= 10.2.3.0.a, 10.2.3.11.k #

p218 12.4."up" # t, t => t #

p224 12.5.MONAD+3 # style TALLY MONAD => style TALLY monad #