ISSN 0084-6198

Algol Bulletin no. 43

DECEMBER 1978

CONTENTS PAGE

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 Robert B. K. Dewar, Courant Institute).

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. Mailing in N. America is handled by the AFIPS office in New York.

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 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 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,
United Kingdom.

Back numbers, when available, will be sent at $3 each. However, it is regretted that only AB32, AB34, AB35, AB38, AB39, AB41 and AB42 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.

AB43.0 EDITOR'S NOTES

The WG met at Jablonna, near Warsaw, at the end of August, and continued its discussion of the "ABSTRACTO" theme. Two of the papers in this issue (those by Sintzoff and Boom) formed part of that discussion, and I hope to publish more of it in the next issue. The discussions will continue at the next meeting, which is to be held somewhere in New Jersey, just before Easter.

Two other outcomes of that meeting were the appearance of the Modules and Separate Compilation proposal in its final form, and the approval of another set of Commentaries on the Revised Report, this time tying up most of the loose ends in the Transput Both of these items apear in this issue (copies of the Modules proposal can also be obtained from the Mathematisch Centrum, Amsterdam). The main effort now proceeding on the ALGOL 68 front is the preparation of a Model Implementation of the Transput, which it is hoped to publish about the middle of next year.

This issue of the ALGOL Bulletin is the thickest ever, due mainly to the inclusion of the two documents mentioned above. As a result, there are a few items which have had to be held over to the next issue, which should therefore follow fairly soon.

AB43.1 Announcements

AB43.1.1 A Finite State Lexical Analyzer for the Standard Hardware Representation of ALGOL 68, by H. B. M. Jonkers

Abstract.

A finite state lexical analyzer for ALGOL 68 programs written in the standard hardware representation is described. The analyzer is written in a very simple language, allowing semi-mechanical translation to an arbitrary language. The whole language, including format-texts, is dealt with.

This Report will be published in full in the next issue of AB. In the meantime, it is available as Report IW 98/78 from the Mathematisch Centrum, 2e Boerhaavestraat 49, Amsterdam.

AB43.1.2 Book Review.

ALGOL 68 - A first and second course, by Andrew D. McGettrick

This book arose out of a course at Strathclyde University to teach ALGOL 68 over a period of 2 years. Teaching ALGOL 68 is the operative word, since it is in no way intended to teach proramming as an art in itself - in this it may be contrasted with that other book from Strathclyde, Programming and Problem Solving in ALGOL 68 by A. J. T. Colin, which teaches programming and just hapens to use ALGOL 68 as a suitable tool (see AB42.1.6).

However, it would be a brave student who succeeded in learning the language by reading this book from cover to cover, for there is much wood to obscure the trees. Technically, it is almost perfect. The whole language is covered. No detail is left out. Every sentence in it is true - but at what a price, for every sntence is prefixed by a long list of conditions which are necessary for its truth, and postfixed with a list of exceptions which are to be discussed at length in some later chapter. "Pedantic", I think, is the word for it, although the style is applied so uniformly throughout (with nary a quotation to enliven it) that perhaps a better word would be "dour" (which is a Scottish technical term).

However, if you know more or less what is in the language and want to enquire about some specific detail, then you will certainly find it here (and with an excellent index to lead you to the right spot). Suppose you want to know all there is to know about recursive mode-declarations, say. Fine! Turn to the part of the chapter concerned. There it will tell you how to construct one. Then follow two pages to tell you which modes are well formed, including all the well-known impossible examples. After that, there is a whole page devoted to mode-eqivalence. If you have struggled through so far (and skipped over a page on the use of flexible names, which now intervenes), you will come at last to what you were really looking for, namely a description of what these strange modes are meant to be used for, the construction of lists and trees. But you then have to go to the exercises at the end of the chapter to find any worthwhile tree examples. The whole book is like this. Once he starts discussing some particular topic, he proceeds to discuss the whole of it. I don't think Dr McGettrick intended to write a work of reference, but that is what he has done. If you are looking for a book to teach the full language, therefore, you must look to Pagan (A Practical Guide to ALGOL 68 - see AB40.1.7) but, contrariwise, you should not look to Pagan to solve obscure technical points.

There are exercises at the end of each chapter (with answers at the back), but one could have wished for more extended examples in the text. An appendix contains standard-prelude tables and the syntax chart from AB37.4.7.

C. H. Lindsey

AB43.1.3 <u>Syntax diagrams for ALGOL 60 and SIMULA 67</u>

A booklet entitled "ALGOL60 / SIMULA67 Syntaxdiagramme", by H. Engelke and B. Kalhoff (Bericht 7806, Oktober 1978), can be obtained from the Institut fuer Informatik und Praktische Mathematik, Christian-Albrechts-Universitaet, D-2300 Kiel 1, Olshausenstrasse 40-60, W. Germany. It contains syntax diagrams for ALGOL 60 and SIMULA 67 in a style somewhat reminiscent of those accompanying the PASCAL report. Although the commentary (of which there is not much) is in German, the diagrams themselves are in English.

AB43.1.4 <u>Modified ALGOL 60 - Erratum</u>

In paragraph 4.6.4.2 in both 'A Supplement to the ALGOL 60 Revised Report' (The Computer Journal, Vol. 19, page 281, and also in SIGPLAN Notices , Vol. 12, Number 1, January 1977 , page 58), and 'Modified Report on the Algorithmic Language ALGOL 60' (The Computer Journal, Vol. 19, page 371):

(i) the label and colon, <GAMMA symbol>: should be lowered by one line, to appear before if instead of before <THETA symbol> ;

(ii) an additional statement, and semicolon, <THETA symbol> := B; should be inserted between S; and V := V + <THETA symbol>; .

The reasons for the above Erratum, which has been authorized by the Working Group, are given in The Computer Journal, Vol. 21, page 282.

AB43.1.5 <u>ALGOL 60 Reports</u>

The British Computer Society has recently published a booklet entitled "ALGOL 60" which contains the Revised Report, the Supplement to the Revised Report, the Modified Report, and Notes on the ISO Hardware Representation. Copies of this booklet may be purchased from BCS headquarters, 13 Mansfield Street, London W1M 0BP, U.K. at the price of 60p to BCS members and 90p to non-members. The erratum mentioned above has been incorporated in these texts.

AB43.2.1

UNIVERSITY OF CAMBRIDGE
COMPUTER LABORATORY

Head of Department
Prof. M. V. Wilkes, F.R.S.

Director of the University
Computing Service
Dr. D. F. Hartley

Corn Exchange Street
Cambridge CB2 3QG
Telephone (0223) 62435

Dr. C.H. Lindsey,
AB Editor,
Department of Computer Science,
University of Manchester,
MANCHESTER, M13 9PL.

1978 November 6

Dear Editor,

AB42.4.2 String escapes for Algol 68

1) Newline and newpage:

Is Professor Hansen suggesting (a) that the newline/newpage escape characters should be mapped into some character codes, e.g. in EBCDIC X'15' (NL) and X'0C' (FF), and that otherwise no special action be taken, or (b) that these escape characters should cause newline/newpage action when transput?

If the former, these characters do not necessarily produce newline/newpage on an output device (but they can be useful in data communication links).

If the latter, Algol 68 has been defined as having a record-oriented transput system and therefore newline and newpage are routines rather than characters. It is easy to map a language-based record-oriented transput system onto an operating system's stream-oriented transput system, if this should be necessary, but it is not so easy to map a language-based stream-oriented transput system onto an operating system's record-oriented transput system - for example, it would be necessary to scan all strings on output to see if they contained any characters requiring special action, for instance newline/newpage.

In any case, the character set may not have provision for newline or newpage characters.

I would suggest that newline and newpage escape characters could be used as a machine-dependent optional extension.

A further minor point, / is not an ANSI control character but a FORTRAN FORMAT item; the ANSI control character for newline is the blank (space) character.

2) Space:

Is the intention of the escape 'ƀ -> ƀ (where ƀ represents a space) (a) to cause the effect of calling the space routine and thereby to skip over the next character, or (b) to represent a blank character (because typographical display features are not significant in a program).

If the former, the implementation would need to test every character in an output string to ensure that no overprinting occurred with an escaped space - e.g.

```
    print((newline,"abcd")); setcharnumber(standout,1);
    print("XY' Z")
```

and there may also be problems in the provision of a suitable character code within the character set.

If the latter, there is no advantage with that representation as the escaped character (i.e. the blank following the apostrophe) would be ignored as being a typographical display feature. For this reason, ALGOL68C uses 'S to represent a blank character - however, the ALGOL68C compiler does not ignore blanks within string and character denotations so the ALGOL68C escape for blank is actually of no practical use, and, because strings with non-escaped blanks are easier to read (by humans) than those with escaped blanks, I hope that other compilers will also treat blanks within string denotations as being significant.

3) Character cases:

According to the AB42.4.2 proposal, the string denotation

```
        "ALGOL68C error message"
```

would yield the string

```
         algol68c error message
```

which is a little strange. Instead, I propose (and intend to implement in ALGOL68C) the following mechanism for providing case changes:

'U or 'u	force subsequent characters to upper case
'L or 'l	force subsequent characters to lower case
'R or 'r	subsequent characters are in the case that they are written

Thus the denotations

```
        "'UALGOL68C 'LERROR MESSAGE"
        "'ualgol68c 'lerror message"
        "'ualgol68c 'rerror message"
        "ALGOL68C error message"
```

all produce the string

```
         ALGOL68C error message
```

and the first two forms are invariant over any change of case of the written characters. The forcing action of 'U and 'L would of course terminate at the end of the denotation and I suggest that it should also terminate at a string break, but I do not have strong feelings on that particular point.

Yours sincerely,

Chris Cheney

C.J. Cheney

AB43.3.1

Commentaries on the Revised Report

The following commentaries are issued by the Sub-committee on ALGOL 68 Support, a standing sub-committee of IFIP WG 2.1. They deal with problems which have been raised in connection with the Revised Report on the Algorithmic Language ALGOL 68, and mostly take the form of advice to implementers as to what action they should take in connection with those problems. These commentaries are not to be construed as modifications to the text of the Revised Report.

Note that commentaries are not being published on trivial misprints. Those concerned about such misprints (and especially those preparing new printings of the Report) should apply to the Editor of the ALGOL Bulletin for the latest list of agreed Errata.

{{The first two commentaries below have already been published in AB42.3.1. They are reprinted here for the sake of completeness.}}

1) Interruption of loops.

Although the semantics of 3.5.2 suggest that a count of the number of iterations of a loop should be kept even when the for-part and the intervals of a loop-clause are EMPTY, it is clearly unnecessary for an implementation actually to implement the count in this case, and it would therefore be unreasonable for an implementation to interrupt (2.1.4.3.h) the elaboration simply because such a count had overflowed. Thus, the elaboration of WHILE TRUE DO SKIP OD would be expected to continue beyond maxint iterations and would not be terminated unless some other action of the operator or operating system intervened.

On the other hand, if a for-part or a FROBYT-part is present in a loop-clause an iteration count must be kept and will be subject to the arithmetic limitations of the hardware. If this count should overflow, therefore, it is reasonable for the implementation to interrupt the elaboration under the provisions of 2.1.4.3.h. For example, in:

```
    FOR i FROM maxint-3 TO maxint DO print(i) OD
```

the implementation may attempt to compute the quantity (maxint+1) (as is indeed suggested by 3.5.2.Step 4), and it will then be quite justified in interrupting.

2) Plus operator on strings.

The + operator for STRINGs declared in 10.2.3.10.i works with strings whose descriptors are exactly flat (2.1.3.4.c), e.g.:

```
    LOC [1:0] CHAR + "abc"    # yields "abc" #
```

but has undefined semantics if a descriptor is "super flat", e.g.:

```
    LOC [1:-1] CHAR + "abc"   # should have yielded "abc" also #
```

This is an error in the Report, and implementations should accept all such STRINGs and yield the same result as if LOC [1:0] CHAR had been provided.

3) Scope of heap variables.

There is an error in 5.2.3.2.a of the Report. The intention of this section was to ensure that the scope of the heap should be the same as that of the outer level of the particular-program so that, for example, the following would be well defined:

```
BEGIN
MODE PCHAIN = STRUCT(PROC(INT)VOID p, INT i, REF PCHAIN next);
LOC REF PCHAIN pstart := NIL;
LOC INT j := 0;
PROC p = (INT a)VOID: j +:= a;
    # the scope of p is determined by its use of the variable j #
...
pstart := HEAP PCHAIN := (p, 1, pstart);
...
END
```

In fact, the first environ created during the elaboration of the particular-program contains 4 other environs (including the primal one) nested within itself {2.2.2.a and 10.1.1}. Therefore, to achieve the intended effect, the first requirement of Case B of 5.2.3.2.a should have been

(i) the primal environ {2.2.2.a} is the environ of the environ of the environ of the environ of E1 {sic},

{{The following group of commentaries is concerned with the Transput section of the Report. The items numbered 4 through 16 describe admitted errors in the Report. Items 17 through 21 are concerned with errors in or amplifications of the pragmatics. Items 22 through 30 contain interpretations of doubtful points or suggestions for sub- or superlanguage features.}}

4) Syntactic errors in the standard-prelude.

There are a few cases in the standard-prelude of syntactically incorrect ALGOL 68 (as augmented by 10.1.3). They are as follows:

a) 10.2.3.3.n+1
"(INT r =" should have been "($\underline{L}$ INT r =".

b) 10.3.1.3.cc."on line end"."Example"+3
"PROC(REF FILE file) BOOL:" should have been
"(REF FILE file) BOOL:".

c) 10.3.2.1.d+8
"fixed (SIGN" should have been "fixed ($\underline{K}$ SIGN".

d) 10.3.2.1.j+12
"$\underline{L}$ real width" should have been "L real width".

10.3.2.1.j+23
"string to L int" should have been "string to int".

10.3.2.1.j+26,+28
"$\underline{L}$ max real" should have been "L max real".

e) 10.3.2.1.n+2
"'1.0' and of '1.0 +" should have been
"'$\underline{L}$ 1.0' and of '$\underline{L}$ 1.0 +".

f) 10.3.3.2.a."OP !"+9
10.3.3.2.b+17
10.3.5.2.b+13
The series commencing on these lines were intended to yield the values (CHARs) of their conditional-clauses, whilst ensuring that "read mood" remained set. They should therefore have been of the form:

```
    CHAR cc = IF ... THEN
                ...
              ELSE ...
              FI;
    set read mood (f); cc
```

g) 10.3.5.1.a."edit L compl"-6
"OR NOT sign2 AND exp < 0" should have been omitted.

5) Standconv and FILE.

Although the Report defines the mode of "standconv" (10.3.1.2.d) to be PROC(CHANNEL)PROC(REF BOOK)CONV, and the mode FILE (10.3.1.3.a) to be a very specific structure (albeit with hidden field-selectors), so that the user may in theory (but without practical use) write

```
   standconv (standin channel) (SKIP)
```

and

```
   FILE f :=
      (SKIP, SKIP, SKIP, SKIP, SKIP, SKIP, SKIP, SKIP, SKIP, SKIP,
       SKIP, SKIP, SKIP, SKIP, SKIP, SKIP, SKIP, SKIP, SKIP, SKIP) ,
```

implementers should feel under no obligation to implement "standconv" and FILE with these particular modes (provided, of course, that "standconv" has one CHANNEL parameter) nor to accept phrases such as the above which rely upon them. (See also the footnote on page 262 of the Revised Informal Introduction.)

6) Establish.

The procedure "establish" (10.3.1.4.b) attempts to check the validity of its parameters "p", "l" and "c". There is an error in the Report whereby, given maxpos OF chan = (10, 50, 100), it accepts for (p, l, c) such clearly unsuitable values as (5, 25, 200) and (5, -100, 0). In lines 10.3.1.4.b+8,+9 the operator BEYOND should have been an operator EXCEEDS, defined as

```
   PRIO 7 EXCEEDS = 5,
   OP EXCEEDS = (POS a, b) BOOL:
      p OF a > p OF b OR l OF a > l OF b OR c OF a > c OF b;
```

7) Associate.

There is a bounds error in "associate" (10.3.1.4.e) if the descriptor of the formal-parameter "sss" is flat {2.1.3.4.c} {there are zero pages} or, otherwise, if the descriptor of "sss[1]" is flat {the pages have zero lines}. It is nevertheless intended that such values of "sss" be accepted {the rest of the transput routines still having well-defined meanings in such cases}.

Note also that the lower bounds of all descriptors involved in "sss" are required to be 1, a fact not made clear in 10.3.1.4.ee.

8) Reset.

There should have been a "DOWN bfileprotect; UP gremlins" before the final FI of 10.3.1.6.j. This then ensures that the semantics of "reset" resemble closely those of "close" followed immediately by re-"open". In particular, it makes it clear that implementations may always entrust complete lines of books to their operating systems as they are produced and that alterations made to those lines by the operating system (e.g. padding with blanks to some multiple of the word length, or moving the logical end - which might have been at the end of some line containing text - to the start of the following line) are not in violation of the Report.

9) Whole, fixed and float.

It was intended that the procedures "whole", "fixed" and "float" (10.3.2.1), when provided with combinations of "width", "after" and "exp" parameters under which no number could possibly be converted (e.g. "whole(x, +1)" and "float(x, 8, 4, 3)") should call "undefined" before possibly returning "errorchar"s.

There is an error in "whole" whereby "undefined" is not called in the case of "whole(1, +1)" (nor in the case of "float(1, +7, 3, +1)"). Line 10.3.2.1.b+11 should have been

```
   IF length = 0 THEN undefined; ABS width * errorchar
   ELIF char in string (errorchar, LOC INT, s)
```

Contrariwise, "fixed(-.123, -4, 3)" does call "undefined" even though "fixed(+.123, -4, 3)" returns, quite correctly, ".123". The test "length > after" in 10.3.2.1.c+4 should not have depended on the sign of "x".

Likewise, "float(1\100, +6, 1, +2)" calls "undefined", even though "float(1\-4, +6, 1, +2)" returns, quite correctly, "+.1\-3". If the failure of the tests on "before" and "after" at the beginning of "float" were to call "undefined" only on the initial call of "float", and were to return "errorchar"s from within its recursive calls, then this example would return "errorchar"s without calling "undefined" first. This is more reasonable, and implementations should behave accordingly.

If "width" = 0, it is intended that the shortest possible string containing the correct result should be returned (except in "float" where the "width" parameter should never be zero). Following this intention, if a string of "errorchar"s is to be returned a string of length 1 would be appropriate, rather than the empty string at present prescribed. The only cases affected are those such as "fixed(x, 0, -3)" and "float(x, 0, -3, 3)", both of which will have first called "undefined".

In "fixed", line 10.3.2.1.c+8, use is made of integer arithmetic in the form of "L 10 ↑ length". This might overflow with large values of "y". Such was not the intention, and real arithmetic, in the form of "L 10.0 ↑ length", should have been used.

In "fixed", line 10.3.2.1.c+14, the test "y < L 1.0" is intended to yield TRUE if there are no significant digits before the decimal point. Due to the rounding in the preceding call of "subfixed" (as in "fixed(0.996, -5, 2)") it may yield TRUE erroneously. The test should have been

```
   y + L .5 * L .1 ↑ after < L 1.0
```

In "float", if "before" = 0 and "width" < 0 and "v" > 0, "⊥" should be yielded before the decimal point. There is an error whereby "0" is yielded in this position. One possible cure is to replace the assignation commencing in 10.3.2.1.d+7 by

```
   s := (x < L 0.0 | "-" |: width > 0 | "+" | "⊥")
      + fixed (y, -(ABS width - ABS exp - 2), after)
      + "\" + whole (p, exp);
```

For examples of the use of these procedures see Commentary 18, and for a discussion of accuracy and the properties of real arithmetic as used in them see Commentary 25.

10) Putting of strings.

"put("")" at end of page has no effect (i.e. no event is provoked). It should have been ensured that the current line was "good", so that the empty string could have been considered to have been written upon it. Therefore, before the loop-clause on line 10.3.3.1.a+39, "(NOT get good page (f, FALSE) | undefined)" should have been present, and likewise "(NOT get good page (f, TRUE) | undefined)" before the loop-clause on line 10.3.3.2.a+77.

11) Put char.

There is an error in "put char" (10.3.3.1.b) whereby the "char error mended" routine called in line 10.3.3.1.b+33 may, by performing a read operation, set the read mood. There should therefore have been a call "set write mood (f)" before the call "check pos (f)" in line b+36.

12) Get next picture.

If a collection is replicated zero times, as in "$ 1 n(0) (3d, 3d) 1 $", its pictures should not be selected {10.3.4.1.1.gg} although the insertions immediately preceding and following it should be performed. The procedure "get next picture" (10.3.5.b), as presently formulated, erroneously selects the pictures of the collection once.

Also, in line 10.3.5.b+22 of "get next picture" there is a test to ensure that "undefined" is called if the staticizing of the insertion or the elaboration of the replicator should cause a different format to be associated with the file (for there is no sensible continuation in such a case). The same test should have been made again after line 10.3.5.b+23, in case the getting or putting of the insertion should have associated a new format.

Likewise, to ensure uniformity of treatment of format-patterns and collections, a similar test ought to have been made after lines 10.3.5.j+4 and 10.3.5.j+5 of "do fpattern".

13) Do fpattern.

The assignation "cp OF aleph[forp] := 0" in 10.3.5.j+12 will cause an unintended index overflow in a subsequent call of "get next picture". It should have been "cp OF aleph[forp] := 1" {cf. 10.3.5.b+9}.

14) Putf.

In 10.3.5.1.a."edit L compl"-4, the expression "t[:b] + point + t[b+2:]" assembles an incorrect string causing, for example, "printf(($ 4d.d $, 13.2))" to print "013.2." instead of "0013.2" and "printf(($ 5d.d $, 13.2))" to be undefined {note that "subfixed" suppresses all nonsignificant leading zeroes}. The expression should have been "(a = 0 | t + point | t)".

15) Edit string.

The declaration "CHAR sj = s[j]" in line 10.3.5.1.b+11 results in a bounds error in cases such as "printf(($ 3z+n(0)d $, 0))". {"$ 3z+ $" would also have caused the problem had it been syntactically legal. "edit string" is asked to edit "+000" with, effectively, the sequence of markers "u,u,u,+". The first "u" is matched against the "+", causing "again" to be set to TRUE and "j" to be incremented so as to point one character further along the string than expected. This is still true when the "+" marker is reached, hence the bounds error.} A sufficient, but not too elegant, solution is to make that declaration

```
   CHAR sj = (j ≤ UPB s ¦ s[j] ¦ "0")
```

16) Indit string.

In "indit string" (10.3.5.2.b), in the case when the marker is "a" or "b" with a replicator of value zero, as in "readf(($ n(0)a $, s))", "no sign" is never set to TRUE, so that the example erroneously assigns "+" to "s".

A satisfactory cure would be to replace 10.3.5.2.b+21 by

```
   s := (marker OF sf[1] = "b" OR marker OF sf[1] = "a" ¦ "" ¦ "+");
```

and to remove the declaration of "no sign" and all statements of which it is a constituent.

Also, the flag "space found" should have been reset to FALSE after an "e" marker or an "i" marker (lines 10.3.5.2.b."marker="e""+2 and "marker="i""+2) so as to prevent, for example, "readf(($ 3z-d e z-d $, x))" from erroneously accepting "␣1234\123".

17) Oversimplified pragmatics.

The Report contains examples of pragmatic remarks which, whilst they indicate the intention of the formal definition so far as most practical situations are concerned, nevertheless, if examined legalistically, do not tell the exact truth. It should be emphasized that the pragmatics were never intended to be an alternative definition of the language. However, it has been thought useful to document some of these cases.

a) 10.3.1.3.dd.last sentence
   A conversion key may also be the "standconv" of some channel, as in
   make conv(standout, standconv(standin channel))

b) 10.3.3.second sentence
   A PROC(REF FILE)VOID routine provided by the user may also be a data-list element.

c) 10.3.3.2.aa,bb,cc,dd
   In all of these, the phrase "the book is searched from the current position" would be more accurate.

d) 10.3.3.2.hh
   Before any characters are read, "newpage" will be called if necessary.

18) Examples of whole, fixed and float.

The examples of the use of "whole", "fixed" and "float" in 10.3.2.1 do not illustrate all of the possibilities of those procedures. Here is a more complete set of examples:

print (whole (i, -4))
   which might print "␣␣␣0", "␣␣99", "␣999", "9999", "␣␣-9", "␣-99", "-999" or, if i were greater than 9999 or less than -999, "****", where "*" is the yield of errorchar;
print (whole (i, +4))
   which might print "␣␣+0", "␣+99", "+999", "␣␣-9", "␣-99", "-999" or,

```
   if i were greater than 999 or less than -999, "****";
print (whole (i, 0))
   which might print "0", "99", "999", "9999", "99999", "-9", "-99",
   "-999", "-9999" or "-99999";
print (whole (2.3, 0))
   which prints "2";
print (whole (2.7, 0))
   which prints "3";
print (whole (i, -1))
   which might print "9";
print (whole (i, +1))
   which must always call undefined (then possibly returning "*"), since
   there exists no value of i which could be converted in a width of +1;

print (fixed (x, -6, 3))
   which might print "⊥0.272", "⊥2.718", "27.183", "271.83" (in which one
   place after the decimal point has been sacrificed in order to fit the
   number in), "2718.3" (2 places sacrificed), "⊥27183" (all places
   sacrificed), "-0.272", "-2.718", "-27.18" (1 place sacrificed),
   "-271.8", "⊥-2718" or, if x were greater than 999999 or less than
   -99999, "*******";
print (fixed (x, +6, 3))
   which might print "+0.272", "+2.718", "+27.18", "+271.8", "⊥+2718",
   "-0.272", "-2.718", "-27.18", "-271.8" or "⊥-2718;
print (fixed (x, -4, 3))
   which might print ".272";
print (fixed (x, +5, 3))
   which might print "+.272" or "-.272";
print (fixed (x, 0, 3))
   which might print ".272", "2.718", "27.183", "271.828", "-.272",
   "-2.718", "-27.183" or "-271.828";
print (fixed (x, n, 0))
   which must always print the same as print (whole (x, n));
print (fixed (i, -6, 3))
   which might print "⊥0.000", "99.000", "999.00", "-9.000", "-99.00" or
   "-999.0";
print (fixed (x, -6, 6)),
print (fixed (x, +6, 5)),
print (fixed (x, -6, -5))
   all of which must always call undefined (then possibly returning
   "*******"), since there exists no value of x which could be so
   converted or, in the last case, since the after parameter of fixed may
   not be negative;

print (float (x, +9, 3, +2))
   which might print "+2.718\-1", "-2.718\+0", "+2.72\+11" (in which one
   place after the decimal point has been sacrificed in order to fit the
   exponent in), "+2.7\+111" (2 places sacrificed), "+27\+1111" (3 places
   sacrificed), "+3\+11111" (4 places sacrificed) or, if ABS x were
   greater than 9.5\99999, "**********";
print (float (x, -9, 3, -2))
   which might print "⊥2.718\-1", "-2.718\⊥0", "⊥2.718\11",
   "⊥2.72\111", "⊥2.7\1111", "⊥27\11111", "⊥3\111111" or, if ABS x were
   greater than 9.5\999999, "**********";
print (float (x, +9, 4, +2))
   which might print "+.2718\+0" or "-.2718\+1";
print (float (x, +9, 0, +2))
   which might print "+27182\-5" or "-27182\-4";
print (float (0.0, +9, 2, +2))
   which prints "⊥+0.00\+0";
print (float (x, m, n, 0))
   which must always print the same as print (float (x, m, n-1, -1)) if
   n>0, or the same as print (float (x, m, n, -1)) if n=0;
print (float (x, 0, m, n)),
print (float (x, +7, 3, +2)),
print (float (x, 4, 0, +2)),
print (float (x, 9, -3, 2))
   all of which must always call undefined.
```

19) Complementarity of put and get.

Generally speaking, if a value of some mode A is successfully output using "put" at a given position (p, l, c) in the book, it may be reinput into a REF A name by a call of "get" at that same position. Moreover, the current position after the "get" will be the same as it was after the original "put".

The principal exceptions to this are listed below. It is assumed that the same CONV is in use on both occasions and that no event routines have been provided. It is also assumed that the error reported in Commentary 10 above has been corrected.

a) If a string that is put includes characters in the "term" field of the file, the reinput string will be truncated to just before the first such character.

b) If a string that is put does not completely fill the line, and if the file is not compressible or another string is subsequently put on the same line, the reinput string will extend to the end of the line (or to an earlier terminating character). As a special case of this, putting an empty string will not result in reinputting an empty string unless the line had already overflowed (10.3.1.5), or "newline" was immediately called (the file being compressible) or a terminating character was immediately put.

c) If a string that is put is split over the end of a line (the default action when no "line end" routine is provided), the reinput string will be truncated to the line end. As a special case of this, if a non-empty string is put when the line has overflowed, an empty string will always be reinput.

d) If the mapping performed by the CONV is not the same in both directions, then differences may arise. For example, it is quite possible that "A" and "a" both map into the external character "A", which would presumably be reinput as "A".

e) If, when putting, the current position is before the logical end of file (which with a sequential access book can only happen if the logical end is within the current line, but with a random access book can happen anywhere), the characters already present between the current position and the logical end can affect what is read back. For example, supposing that "int width" = 6:

```
   put (f, (newline, "␣␣␣␣␣␣␣456␣"));
   set char number (f, 1);
   put (f, 123)
```
will result in the printed line
```
   "␣␣␣+123456␣"
```
with obvious disastrous consequences upon a subsequent "get (f, i)".

A related problem arises if there is insufficient room for an arithmetic value on the remainder of the current line. Suppose that the length of a line is 10 characters, that "int width" = 6 as before, and that "real width + exp width + 4" > 6:

```
   put (f, (newline, "␣␣␣", 123456));
   set char number (f, 4);
   put (f, 2.718281828)
```

The 2.71828, in some form such as +2.718281828\␣+0, will be put on the next line so that a subsequent "get (f, x)", at the position of the original "put", will obtain the real number 123456.0.

20) Putting of numbers.

In the pragmatic remarks 10.3.3.1.aa, bb, cc, after "the current line", it would be helpful to insert "(or one less than this when at the start of the line)".

21) General patterns.

The pragmatic remark 10.3.4.10.1.bb is incomplete. It was intended to read:

```
   bb) A value ... whose pattern Q ...:
.  P is staticized;
.  the insertion of Q is performed;
.  (any parameters ...) ... using get;
.  the insertion of P is performed.}
```

22) CONVs.

The CONV feature of the transput is one of the less satisfactory features of the Report. It does not interface well with conversion features that are likely to be provided by the hardware or operating system (which would be better invoked by JCL commands or features in "idf" strings, anyway). Its correct implementation is difficult to do efficiently.

Noting that the Report nowhere obliges an implementer to provide any CONVs beyond one "standconv" for each CHANNEL (and these are likely to be null operations, or at least very straightforward, in most systems), we wish to recommend that implementers should not in general provide additional CONVs unless they have pressing problems with their local character codes.

In any event, if any additional CONV is provided, at least the digits and the other characters required for the transput of arithmetic and BITS values should always be convertible.

23) Physical file end.

When (on output) the physical end of the book is reached, the event routine corresponding to "on physical file end" is supposed to be called as soon as the user attempts to put any character on the line (just beyond the physical end) that isn't there. If the physical limits of the book are known to the run-time system in advance (e.g. they are simply the "p", "l" and "c" parameters of "establish"), this raises no problem. In practice, however, the limit usually becomes known only when the operating system refuses to accept the contents of the buffer, which will not occur until, having put characters on the line with apparent success, the user calls, for example, "newline".

Although this situation is not recognised by the Report, it is nevertheless recommended that the physical-file-end event routine should thereupon be called. In terms of the transput model of the Report, it should be as if some system-task had suddenly reduced the size of the text of the book just as the body of "newline" (or "newpage") was entered, causing "physical file ended (file)" {10.3.1.5.h} to return TRUE.

Users should be warned that any characters written to the buffer up to that point may have been lost. They can test whether this has happened by seeing whether "char number (file)" {10.3.1.5.a} returns an integer greater than 1 (note that in all present circumstances where the physical-file-end event is called the char number will be found to be 1, indicating an empty line).

24) Reidf.

The procedure "reidf" can only be called when the book to be renamed has already been opened via some file. However, not all operating systems are able to perform their renaming function on an opened book. For implementers faced with such systems, one of the following courses of action is recommended:

a) Do not provide the "reidf" facility at all (the Report nowhere requires that "reidf possible" should ever return TRUE);

b) "reidf" causes no immediate action, but the revised "idf" is remembered and the book is renamed when the file is eventually "close"d (or "lock"ed).

It is to be noted that, where "reidf" is provided, the set of strings acceptable as its "idf" parameter may be only a subset of the "idf"s acceptable to "open" and/or "establish", especially if the latter are permitted to contain information concerning the disposal of, or the access rights to, the book.

25) L int width, etc.

The environment enquiries "L int width", "L real width" and "L exp width" (10.3.2.1.m, n, o) are primarily provided in order to fix the numbers of digits to be allowed for when numbers are output with "put". In the case of REAL (and hence also COMPL) numbers, a sensible choice of "L real width" will depend upon how accurately the implementation actually performs its conversions in "put". "L exp width" should reflect the manner in which "L max real" is actually converted.

A numerical analyst would expect, in an ideal world, that conversions would produce sufficient digits to ensure that different real numbers are always converted to different strings. In actual implementations this may not be practicable. In any case the value of "L real width" should reflect what is actually done in "put".

The Report does not seek to define the accuracy of the conversions performed by "fixed" and "float" (10.3.2.1.c, d) (and hence by "put" and "putf") and by "string to L real" (10.3.2.1.j). Here, as elsewhere where operations are performed upon real numbers {2.1.3.1.e}, implementations should adhere to the best practices of numerical analysis. It is best, therefore, not to regard the texts of these procedures as defining the strings or values which would have been obtained had the real-number operations therein actually been performed according to the limitations of the particular implementation, but rather to consider the strings or values which would have resulted if real numbers and operations in the sense of mathematics had been used, and then to consider how close to these ideal results it is reasonable for the implementation to get. In particular, no question of arithmetic overflow should arise (so that it is defined that all values up to and including "L max real" may be converted).

Although, with this interpretation, the precise definitions of these environment enquiries become of less importance, it should be noted that there are some specific errors in "L int width" and "L real width":

a) In the present definition of "L int width", "maxint = 100" implies "int width = 2". Also, real arithmetic is used which is undesirable where an exact result should be obtained. Line 10.3.2.1.m+4 should therefore have been

```
   WHILE L̲ 10 ↑ (c-1) ≤ L max int % L̲ 10 DO c +:= 1 OD
```

b) The comment in 10.3.2.1.n should have been:
"... different strings are produced by conversion of 'x * L̲ 1.0' and of 'x * (L̲ 1.0 + L small real)', for all admissible, non-zero x, ...".

26) INTYPE.

Due to an oversight in the definition of INTYPE (10.3.2.2.d), it is not possible to write

```
   LOC STRUCT (BOOL b, STRING s) x; read (x)
```

although the given straightening algorithm handles this case correctly (as will all likely implementations); moreover, "print (x)" is allowed.

Implementers are therefore recommended to allow, in addition to the 'MOOD's from which INTYPE is presently united, modes which (whilst still subject to the other restrictions given) contain 'flexible row of character' {but not other 'flexible' modes}.

27) Default action of event routines.

It is generally the case that, in each situation where an event routine is called, a default action is provided (possibly a call of "undefined") and is taken if the routine returns FALSE (even if it has actually mended the situation). There are, however, three exceptions to this, brought about when the position is not "good" {10.3.1.6.dd}: viz. when a string is being input with "get", or any value is being input with "getf", or any value is being output with "putf". In these cases, the default action (to terminate the input in the first case, and to call "undefined" in the others) is only taken if the position remains "bad", and the value returned by the event routine makes no difference, except insofar as it may prevent any further attempts to fix the situation (see "check pos" {10.3.3.2.c}).

To obtain a more consistent treatment of event routines, implementers are recommended always to take the default action when an event routine returns FALSE. This means that in "get", lines 10.3.3.2.a."(REF STRING ss)"+2:+3 are to be interpreted as if they had been written:

```
   WHILE
      IF NOT check pos (f)
```

In "edit string", line 10.3.5.1.b+3 is to be interpreted as if it had been

```
   (NOT supp | (check pos (f) | put char (f, c) | undefined));
```

with similar changes to be made in 10.3.3.1.c-4 ("put char"), 10.3.5.g+10 ("put insertion") and 10.3.5.i+14 ("alignment"), and in "indit string", line 10.3.5.2.b."OP !"+4 is to be interpreted as if it had been

```
   IF CHAR k; (check pos (f) | get char (f, k) | undefined);
```

with similar changes to be made in 10.3.5.h+11 ("get insertion") and 10.3.5.i+10 ("alignment").

To accomodate all these changes, "check pos" (10.3.3.2.c) has to be rewritten as follows:

```
   PROC ? check pos = (REF FILE f) BOOL:
      BEGIN
         BOOL reading = read mood OF f,
         BOOL ended := TRUE;
         WHILE IF get good page (f, reading)
            THEN (line ended (f) | (line mended OF f)(f)
                                 | ended := FALSE)
            ELSE FALSE
            FI
         set mood (f, reading);
         NOT ended
      END ;
```

28) Last insertion of a collection.

Normally, when a picture is encountered during formatted transput, it is "staticized", i.e. its dynamic replicators are all elaborated (collaterally) before transput using that picture commences. Moreover, if the transput is terminated by a jump at this stage, any subsequent transput will start with the next following picture. Clearly, it is not appropriate to staticize the whole of a collection-list at one time, and it is therefore prescribed that its pictures be staticized one by one as they are required for transput. Finally, at the end of the collection-list, there may be a final insertion to be performed when the collection-list has been repeated as required by its replicator. The Report defines that the replicator of this final insertion be elaborated collaterally with the staticizing of the last picture on its last time round; i.e. _before_ any possible side effects of the transput using that picture.

This is not what a user might expect nor is it easy to implement, and implementers should therefore feel free to elaborate it after the transput of the final picture (but before returning from "putf" or "getf", of course). If the transput should terminate at this point, the collection-list should be deemed to have been completed and future transput should start with the next complete picture or collection-list.

The final insertion of a format-patern should be treated similarly.

29) G patterns.

In formatted output a g-pattern can be used to obtain the effect of "put" and, if it is provided with parameters, of "whole", "fixed" and "float" also. Thus "printf (($g(6,3), x))" is exactly equivalent to "print (fixed (x, 6, 3))". On input, a g-pattern can give the effect of "get" but, since "whole", "fixed" and "float" cannot be used with "get", there is no meaning for any parameters, and none should therefore be present.

In fact, the Report gives an entirely false impression of input/output compatibility by permitting parametrized g-patterns on input, but directing that their parameters be ignored. It would therefore be better to regard such parametrized g-patterns on input as not being "input compatible" {10.3.4.1.1.ii} with their data-list values and to call the "on value error" event (with default action "undefined" if it returned _false_) whenever one was encountered.

30) Binary transput.

An implementer of the binary transput defined by the Report is not, in general, able to output stored values in the form in which they are stored internally (which may include packing, storing of fields in different orders and inclusion of extra fields such as garbage-collector templates). This is because of the requirement to be able subsequently to reinput their component fields or elements either singly, or stowed in other ways. For example, the following is well defined:

```
   putbin (standback,
         (STRUCT (REAL a, [] BOOL b) (2.0, (TRUE, TRUE, TRUE)),
          STRUCT (CHAR c, INT d) ("c", 1)));
   reset(standback);
   getbin(standback,
         (LOC STRUCT (REAL a, BOOL b1),
          LOC STRUCT ([1:2] BOOL b2, CHAR c),
          LOC INT))
```

There exists a sublanguage in which output values of some mode may only be reinput into variables of that same mode. Clearly, this sublanguage is easier (and more efficient) to implement, whilst detracting little from the power of the language (and even increasing its safety), and it is therefore commended to implementers.

The sublanguage is defined by the following modified forms of the procedures in 10.3.6:

```
PROC ? to bin = (REF FILE f, OUTTYPE x) [] CHAR:
   C present text C ;

PROC ? from bin = (REF FILE f, OUTTYPE y, [] CHAR c) OUTTYPE:
   C present text C ;

PROC put bin = (REF FILE f, [] OUTTYPE ot) VOID:
   IF opened OF f THEN
      set bin mood (f); set write mood (f);
      FOR k TO UPB ot
      DO [] CHAR bin = to bin (f, ot[k]);
         FOR i TO UPB bin
         DO
            ...
         OD
      OD
   ELSE undefined
   FI ;

PROC get bin = (REF FILE f, [] INTYPE it) VOID:
   IF opened OF f THEN
      set bin mood (f); set read mood (f);
      FOR k TO UPB it
      DO
         OUTTYPE y = C the value referred to by the yield of ot[k] C;
         [1:UPB (to bin (f, y))] CHAR bin;
         FOR i TO UPB bin
         DO
            ...
         OD;
         C the name yielded by ot[k] C := from bin (f, y, bin)
      OD
   ELSE undefined
   FI ;
```

AB43.3.2

A Modules and Separate Compilation Facility for ALGOL 68.

By C. H. Lindsey (University of Manchester)

and H. J. Boom (Mathematisch Centrum, Amsterdam)

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.

Acknowledgements

These proposals, which have been discussed over a long period of time by the ALGOL 68 Support Sub-committee, owe their origin to the proposals of Schuman [1] and of the Cambridge compiler team [2]. They have been discussed extensively at meetings of the Sub-committee and in correspondence between members of its Task Force on Modules and Separate Compilation. The authors of the present work wish to record their thanks to all those who have contributed in this way, and especially to Dr R. B. K. Dewar of the Courant Institute and Dr A. D. Birrell of Cambridge University.
[1] Schuman, S. A., "Towards Modular Programming in High-Level Languages", ALGOL Bulletin No. 37, July 1974, AB37.4.1.
[2] Bourne, S. R., Birrell, A. D. and Walker, I., ALGOL 68C Reference Manual, 1975.

This document is in three sections:
1. Informal Description of Modules and Separate Compilation, by C. H. Lindsey.
2. Formal Deescription of Modules and Separate Complation, by C. H. Lindsey.
3. Implementation Methods for Modules and Separate Compilation, by H. J. Boom.

Informal description of Modules and Separate Compilation.

1 Separate compilation and protection.

These are two distinct concepts which must nevertheless be considered together in order to make a viable system. "Protection" implies a mechanism, better than classical block structure, for preventing indicators defined in one place from being applied in other places where they shouldn't. "Separate compilation" is a compile-time activity, designed to split large programs into manageable chunks and to provide a library mechanism. The features are independent in that the user should not be forced to use the one in order to gain the benefits of the other. On the other hand, the unit whose contents are to be protected will frequently be also a convenient unit for separate compilation, and therefore the use of the two features together should be as comfortable as possible. This proposal does not attempt to provide an "Abstract data type" facility. The proposed protection and separate compilation mechanisms are orthogonal to the existing ALGOL 68 "concrete" data types.

2 Definition modules.

A definition module can be declared anywhere (but typically in the outer reach, and often compiled separately):

```
MODULE F = DEF LOC STRING s; read(s);
               PUB LOC FILE f; open(f, s, standin channel)
           POSTLUDE
               close(f); print(("file ", s, " closed"))
           FED;
```

and it can be accessed anywhere within its reach

```
LOC STRING message;
ACCESS F (LOC STRING t; get(f, t); message := t[2: ])
         <---------- controlled clause ----------->
<--------------- access-clause ------------------->
```

The effect is to elaborate the body of the definition module, inserting the controlled clause just before the POSTLUDE. From within the controlled clause (which is, in general, an ENCLOSED-clause), the identification mechanism first searches the declarations within itself, then those declared PUBlicly in the module (i.e. 'f', but not 's'), and then those in the reach outside the access-clause. An access-clause can return a value, coercions being passed inside it as with other ENCLOSED-clauses:

```
LOC STRING message :=
    ACCESS F
      (LOC STRING t; PROC prs = REF STRING: (get(f, t); t); prs)
```

Observe the difference between

```
ACCESS A,B ( ... )      and      ACCESS A ACCESS B ( ... )
```

both of which are legal. Of course, the second creates one more scope level than the first, but there could also be a difference of meaning if A happened to PUBlicize another definition module B. Moreover, if both A and B happened to PUBlicize the same identifier, the compiler would report an error in the first case, but not in the second. The first form is therefore to be preferred, especially when A and B are separately compiled modules which know nothing of each other's existence and whose complete list of PUBlications may be unknown to the user.

Definition modules are particularly intended for providing packages whose inner workings can be concealed from their users. It is cutomary at this stage to exhibit a module for implementing a stack:

```
MODULE STACK =
DEF
    INT stacksize = 100;
    LOC [1:stacksize] INT st;
    LOC INT stptr := 0;
    PUB PROC
      push = (INT n)INT:
          ((stptr+:=1)<=stacksize ¦ st[stptr] := n
                ¦ print("stack overflow"); stop) ,
      pop = INT:
          (stptr>0 ¦ st[(stptr-:=1)+1]
                ¦ print("stack underflow"); stop)
POSTLUDE
    (stptr/=0 ¦ print("stack not emptied"); stop)
FED;
```

Now this module may be accessed

```
ACCESS STACK (push(1); push(2); print(push(pop)); pop; pop)
```

Note that ACCESS is to be regarded primarily as a mechanism for permitting PUBlicized indicators to be made visible:

```
ACCESS STACK
    (push(1); push(2);
      (PROC push = C something else C, pop = C something else C;
       push; pop;
       ACCESS STACK (print(push(pop)) # prints 2 # )
      ); pop; pop
    )
```

When ACCESS STACK is encountered at the outer level, it is "invoked", i.e. its body is elaborated up to its POSTLUDE and side effects (in this case the allocation of space for 'st') may occur. The ACCESS STACK at the inner level can see the outer one, there is no fresh invocation and the same STACK is accessed. The postlude is not elaborated until the outer ACCESS is finally completed.

Although it can be contrived that two invocations of a module coexist, this is to be regarded as a most unusual situation. Please do not confuse modules with SIMULA classes. If you want to have more than one stack available there is a proper way to go about it.

```
MODULE STACKS =
DEF
    INT stacksize = 100;
    MODE S = STRUCT ([1:stacksize] INT st, INT stptr);
    PUB MODE STACK = REF S;
    PUB PROC
      newstack = STACK:
          (HEAP S s; stptr OF s := 0; s) ,
      push = (STACK s, INT n)INT:
          (REF INT sp = stptr OF s;
           ((sp+:=1)<=stacksize ¦ (st OF s)[sp] := n
                 ¦ print("stack overflow"); stop) ) ,
        pop = (STACK s)INT:
            (REF INT sp = stptr OF s;
             (sp>0 ¦ (st OF s)[(sp-:=1)+1]
                  ¦ print("stack underflow"); stop) )
  FED;
```

Observe that the postlude is not appropriate in this version, and it has therefore been left out. The user may declare STACK variables for himself but, if he is honest, he will pretend he does not know about the STRUCT with which STACKs are implemented. However, there are no secret modes in ALGOL 68, so a malicious user cannot be prevented from writing duplicate declarers and making his own STACKs. Observe that this particular STACKS module reserves no storage space - and indeed its invocation has no side effects whatsoever.

Invocations are thus shared whenever it can be detected statically that this is possible. Modules may access other modules, but it is still possible to avoid all unnecessary invocations at compile time.

```
    MODULE A = DEF ... FED,
           B = DEF ... FED,
           C = ACCESS A,B DEF ... FED;
             # the PUBlicized declarations of A and B are visible inside  C,
             but are not available to a user of  C  unless  he  specifically
             asks for them #
    ACCESS B,C ( ... )
```

Here (assuming nothing is invoked to start with) B is invoked first. The attempt to access C finds that A and B are needed and it therefore invokes A (the first of them). It then finds that B is already invoked, so just makes the existing invocation accessible inside C. After that, C itself can be invoked and finally the invocations of B and C (but not A) are made available to the inside of the controlled clause. When this has been elaborated, the modules are revoked (i.e. their postludes, if any, are elaborated) in the inverse order of their invocation.

Had it been required that the PUBlicized declarations of B should always be visible to accessors of C, then C could have been declared

```
    MODULE C = ACCESS A, PUB B DEF ... FED
```

whereupon the access-clause ACCESS C (...) would have had the same effect as ACCESS B,C (...) previously (except that the order of invocation would then have been A, B, C instead of B, A, C).

Here is a carefully chosen confusing example to show exactly what happens:

```
    MODULE A = DEF PUB LOC INT i := 0 FED;
    MODULE B = ACCESS A DEF i+:=1; ACCESS A (i+:=1) FED;
    PROC c = VOID: ACCESS A (i+:=1; ACCESS B (print(i)));
    ACCESS A (i+:=1; c)
```

We have, at various times, considered schemes which would have made this example print 1, 2, 3 or 4, but in the version now defined it prints 3. To see why this is so, consider first those access-clauses which will not invoke A afresh because they can identify (as shown by the dotted lines) an existing invocation. This leaves two other access-clauses (one of them in the body of the procedure) which are bound to create new invocations of A whenever they are elaborated. Next consider the identification of the applications of 'i'. Clearly, they all identify the 'LOC INT i' in A, but they do so indirectly via particular invocations of A, as shown by the thick lines.

```
MODULE A = DEF PUB LOC INT i := 0 FED;
MODULE B = ACCESS A # whether this invokes a fresh A depends
                 ↑  upon where B is accessed #
                 | DEF i+:= 1; ACCESS A (i+:=1) FED;
                 |_____|              /↑___|
                 ↑_ _ _ _ _ _ _ _ _ _/

PROC c = VOID: ACCESS A # always a fresh A #
                     ↑ (i+:=1; ACCESS B # this B does not invoke
                     |_|              /    a fresh A #
                     ↑_ _ _ _ _ _ _ _/   (print(i)));
                     ↑______________________|

ACCESS A # always a fresh A #
       (i+:=1; c)
```

By the time the call of 'c' is reached, A will have been invoked and the variable 'i' generated thereby will have been incremented to +1. However, the call of 'c' invokes another A and generates another variable 'i' which soon gets incremented to +1. The ACCESS B invokes B, but it does not invoke a fresh A, and therefore both the 'i's in B identify the same (i.e. the second) 'i', which therefore gets incremented twice more. Finally, the 'i' in 'print(i)' identifies the second 'i' (whose value is now +3) as shown (B is not involved, as it only accesses A privately).

Here is a final example to show how a well known dangerous example can be made safe:

```
BEGIN
C same as Report 11.12 up to and including MODE PAGE C;
MODULE BUFFERS =
DEF [1 : nmb magazine slots] REF PAGE mag;
    INT in := 1, ex := 1;
    SEMA full slots = LEVEL 0, free slots = LEVEL nmb magazine slots,
         in buffer busy = LEVEL 1, out buffer busy = LEVEL 1;
    PUB MODULE
        CRITICALIN =
        DEF PUB REF [] REF PAGE magazine = mag,
                PUB REF INT index = in;
            DOWN free slots; DOWN in buffer busy
        POSTLUDE
            UP full slots; UP in buffer busy
        FED,
        CRITICALOUT = C similarly C
FED;
ACCESS BUFFERS
    BEGIN
    PROC par call = C as Report C;
    PROC producer = (INT i) VOID:
        DO  HEAP PAGE page;
            get (infile[i], page);
            ACCESS CRITICALIN
                (magazine[index] := page;
                 index MODAB nmb magazine slots PLUSAB 1)
        OD;
    PROC consumer = C similarly C;
    PAR ( C as in Report C )
    END
END
```

3 Libraries.

Library procedures should be grouped together into sensible packages. Thus the library-prelude might contain:

```
MODULE MATMODE = DEF PUB MODE MAT =
                         C the standard mode for matrices C
                 FED;
MODULE MATRICES = ACCESS PUB MATMODE
                  DEF
                      C  declares  a  collection  of  PUBlicly  known
                      procedures for matrix handling, which possibly use
                      some secret inner  procedures  and  secret  global
                      variables, hereby initialized C
                  FED;
MODULE VIBRATIONS = ACCESS MATRICES, PUB MATMODE
                  DEF
                      C  declares  a  collection  of  PUBlicly  known
                      procedures  for  analysing  the  oscillations   of
                      structures, which use (but do not  PUBlicize)  the
                      matrix handling procedures PUBlicized by  MATRICES
                      C
                  FED;
MODULE STRESSES = ACCESS MATRICES, PUB MATMODE
                  DEF
                      C  declares  a  collection  of  procedures  for
                      analysing stresses C
                  FED;
```

These four module-declarations would be compiled into the library independently of one another except that, presumably, MATMODE had to be compiled first and MATRICES had to be compiled (or at least have its PUBlic interface compiled) before the remaining two. Observe that accessors of any of them automatically get to see the mode MAT, but users of VIBRATIONS and STRESSES do not thereby get to see MATRICES.

A particular-program can now invoke one, any two or three, or all of them:

```
ACCESS VIBRATIONS, STRESSES
BEGIN
    ... ... ...
        ACCESS MATRICES
            IF ... THEN ... FI;
    ... ... ...
END
```

The closed-clause here appears to be being elaborated inside two modules. Actually, it is being elaborated inside four. What happens is that the system first tries to invoke VIBRATIONS. It finds that, for VIBRATIONS, MATRICES is required and it can see (at compile time) that no invocation of MATRICES exists in the static environment. It therefore invokes MATRICES (which thereby invokes MATMODE by the same mechanism) and after that it invokes VIBRATIONS. It now tries to invoke STRESSES, which also requires MATRICES (and MATMODE), but now it knows that invocations of these already exist, so it can invoke STRESSES immediately. Inside the BEGIN ... END, the PUBlicized declarations of MATMODE, VIBRATIONS and STRESSES (but not those of MATRICES) are available.

When ACCESS MATRICES is encountered, it again knows at compile time that MATRICES is already invoked. The only action required, therefore, is to PUBlicize the declarations of MATRICES within the IF ... FI. Note that this example also illustrates how a particular-program may begin with an ACCESS (an access-clause is an ENCLOSED-clause).

4 Separate compilation using definition modules.

The following example shows how a compiler, in which the first pass has several phases, would be compiled in several packets. The last packet is a particular-program - the rest are module-declarations which are to be gathered into a "user-prelude", which is in effect a private library. Each packet contains an ACCESS, followed by a list of module-calls. It may be useful to regard the standard-prelude (including the particular-prelude) as another module, and to imagine that each of these lists implicitly commences "ACCESS STANDARDPRELUDE".

```
    MODULE COMMUNICATIONAREA =
        DEF ... FED
----------------------------------------
    MODULE PASS1 =
        ACCESS COMMUNICATIONAREA
        DEF ... FED
----------------------------------------
    MODULE PHASE1A =
        ACCESS PASS1
        DEF
            ...
            PUB PROC phase1a = ... ;
            ...
        FED
----------------------------------------
    MODULE PHASE1B =
        ACCESS PASS1
        DEF
            ...
            PUB PROC phase1b = ... ;
            ...
        FED
----------------------------------------
    MODULE PASS2 =
        ACCESS COMMUNICATIONAREA
        DEF
            ...
            PUB PROC pass2 = ... ;
            ...
        FED
----------------------------------------
    ACCESS COMMUNICATIONAREA
        BEGIN
        ACCESS PASS1
            BEGIN
            ACCESS PHASEIA BEGIN ... phase1a ... END;
            ...
            ACCESS PHASE1B BEGIN ... phase1b ... END;
            ...
            END;
        ACCESS PASS2
            BEGIN pass2 END
        END
```

5 Separate compilation using holes.

The system described above essentially permits the building of programs in a bottom-up manner. However, strong opinions have been expressed that top-down building should also be provided. We found it necessary to propose a completely separate mechanism - the hole - for this, since all attempts to make the gap between the prelude and postlude of a definition module do this job proved fruitless.

```
    BEGIN
    C interesting declarations C;
    ...
        ...
        IF ...
        THEN C more interesting declarations C;
            NEST "a"  # this construct is a formal-hole #
        ELSE C yet more declarations C;
            NEST "b"
        FI;
        ...
    ...
    END
-----------------------------------------
    EGG "a" =
        ( C some serial-clause. All the declarations preserved in  the  nest
        at "a" are available here C )
        # this construct is an actual-hole #
-----------------------------------------
    EGG "b" =
        ( ................. )
```

The three packets shown would be compiled in the given order. Clearly, the semantics simply state that the meaning of the collection of packets is the same as that of the particular-program obtained by removing the formal-holes and stuffing the gaps with their matching actual-holes. The string- (or character-) denotations "a" and "b" are hole-indications. Their syntax is quite different from other indications in the language because they do not obey the usual identification rules of other indicators. Indeed they must be unique within the program. Normally, they should be of the form letter followed by letters or digits, but the formal definition allows some flexibility to suit the local operating environment (10.6.2.b) so that implementers can, for example, interpret them as the names of the files where the relevant interface information has been stored.

Holes also provide a mechanism for introducing program segments written in other languages. Suppose, for example, that the implementer has provided means to access FORTRAN subroutines. Then users would be allowed to write declarations such as the following:

```
    PROC(REAL)REAL function = NEST FORTRAN "FUNCTION";
```

The compiler would then know to generate a FORTRAN-style calling sequence at calls of 'function', and the loader would be instructed to find the subroutine FUNCTION in some FORTRAN-style library. The Formal Definition contains an example (5.6.1.g) of what the syntax might permit for this facility.

There are some problems, especially for implementations using the static/dynamic chain method of keeping track of their stack frames, concerning the scope of routine-texts whose bodies contain formal-holes. The scope of such a routine is therefore made to be the smallest possible scope, as if its body had contained identifiers identifying defining occurrences in every range within which it was contained (just in case the actual-hole eventually stuffed were to contain such identifiers). Thus the elaboration of the following is always undefined:

```
    LOC PROC (REAL) REAL pp;
        BEGIN
        LOC REAL x;
        PROC p = (REAL a) REAL: NEST "p";
        pp := p
        END
```

(because the actual-hole stuffed into "p" might contain an application of 'x'). However, it is usually easy to avoid the problem entirely by writing, for example:

```
    PROC(REAL)REAL p = NEST "p";
```

rather than

```
    PROC p = (REAL a)REAL: NEST "p";
```

In addition to stuffing an actual-hole into a formal-hole, several definition-module-packets may be stuffed as well. Thus we can have

```
    EGG "a" = MODULE A = DEF ... FED
----------------------------------------
    EGG "a" = MODULE B = ACCESS A DEF ... FED
----------------------------------------
and finally
    EGG "a" = BEGIN ... ACCESS A,B ( ... ) ... END
```

Presumably, these (or at least their PUBlic interfaces) would have to be compiled in the order given, but to avoid all possibility of confusion there is a restriction that A and B must not be identifiable (neither as module-indications, nor as mode-indications, nor as operators) in the NEST "a" into which these EGGs are to fit. Indeed, it is reasonable to imagine that all the packets in the VIBRATIONS and STRESSES example above had been stuffed into a formal-hole representing the standard-prelude, as if they had been preceded by an implicit 'EGG "standard prelude" ='. (Thus, whether the standard-prelude is to be regarded as a definition module or as a formal-hole is purely a matter of taste - moreover actual implementers are likely in fact to treat it as a special case different from either.)

6 Compilation systems.

A "module-interface" is the document (written in some cryptic notation only understood by the compiler) which conveys information about PUBlicized declarations from a separately compiled definition module to its accessors. A "hole interface" does the same thing between a formal- and an actual-hole. Interfaces are output by the compilation of the packets which define them and may be re-input when compiling packets which require them. Alternatively, a module-interface (produced by a previous compilation of a definition-module-packet) may be "imposed" on a recompilation of that packet, ensuring if possible that the object-module produced is still consistent with that interface. In this way, re-compilation of other packets dependent upon that interface can be avoided. (However, we see no reasonable hope of imposing hole-interfaces.)

7 Order of compilation.

Clearly, a hole must be compiled before its stuffing. Ordinarily, a particular-program or module must be compiled after any separately compiled module which it accesses. However, this order can be varied by using imposed interfaces.

Suppose that a user wishes to have a module A which is to be used by a main program B, but that he wishes to compile (and even partially test) B before A. He therefore writes a skeletal module-declaration A' which contains just enough to fix the interface between A and B. A' is compiled to produce a module-interface a' (presumably this contains, inter alia, offsets for the indicators PUBlicized in A'). B is now written and compiled using a' (moreover the object-module produced for B is aware of the time stamp that was given to a' at its instant of creation). Next, the final version of A is written but, when it is compiled, the module-interface a' is imposed upon it. Clearly, the compiler will abort if A is not "consistent" with a'. Compiler writers should be encouraged to make their definitions of "consistent" as liberal as possible. For example, there should be no difficulty in accepting the offsets fixed in a' even if the corresponding indicators in A turn out to have been declared in a different order. Note that no new interface a is produced. If now A is to be recompiled to mend some bug, and it is hoped to avoid re-compilation of B, then the inferface produced by or imposed upon the previous compilation of A (e.g. a') should be imposed and the compiler will try to produce an object module consistent with it if it possibly can. If it cannot, it will say so, signifying that recompilation of B cannot now be avoided.

Of course, the user should be aware that he may gain in efficiency, or in improved optimizations, or in the reduction of wasted space, if he finally recompiles A to produce its best interface a, and then re-compiles B using a.

8 Formal definition.

The formal definition of these proposals which follows uses the existing formalism and conventions of the Revised Report. Note that, although it is expressed as modifications to the Report, no authority to alter the official Report text is implied. Moreover, these particular modifications have been chosen so as to minimize the number of places in the Report affected, and had these features been part of the language from the very beginning, their formal definition might have been simpler.

Formal Definition of Modules and Separate Compilation.

Part I - Definition Modules.

{{Module-declarations are new kinds of declarations. New kinds of entry in the nest are therefore needed.}}

```
1.2.3.
  B)  LAYER :: new DECSETY LABSETY INKSETY.
  E)  DEC :: ... ; MOD.
  L)  MODSETY :: MODS ; EMPTY.
  M)  MODS :: MOD ; MODS MOD.
  N)  MOD :: module REVS TAB.
  O)  REVSETY :: REVS ; EMPTY.
  P)  REVS :: REV ; REVS REV.
  Q)  REV :: TAU reveals DECSETY INKS.
  R)  TAU :: MU.
  S)  INKSETY :: INKS ; EMPTY.
  T)  INKS :: INK ; INKS INK.
  U)  INK :: invoked TAU.
```

```
4.8.1.
  E)  PROP :: ... ; INK.
  F)  QUALITY :: ... ; module REVS ; invoked.
  G)  TAX :: ... ; TAU.
```

{{'MOD's will be introduced into the nest by module-declarations. 'INK's will be introduced by module-calls.}}

{{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 MODS{41a,e} :
      module{94d} token,
        NEST1 module joined definition of MODS{41b,c}.
b)  NEST1 module definition of module REVSETY REV TAB{41c} :
      where <REV> is <TAU reveals DECSETY invoked TAU>
          and <TAB> is <bold TAG>,
        where <NEST1> is <NOTION1 invoked TAU NOTETY2>,
        unless <NOTION1 NOTETY2> contains <invoked TAU>,
        module REVSETY REV NEST1 defining module indication
          with TAB{48a},
        is defined as{94d} token,
        NEST1 module text publishing REVSETY REV defining LAYER{c,-}.
```

```
c)  NEST1 module text
          publishing REVSETY TAU reveals DECSETY INKSETY INK
          defining new DECSETY1 DECSETY INK{b} :
      where <INKSETY> is <EMPTY> and <REVSETY> is <EMPTY>,
        def{94d} token,
        NEST1 new new DECSETY1 DECSETY INK module series
          with DECSETY without DECSETY1{d},
        fed{94d} token ;
      NEST1 revelation publishing REVSETY defining LAYER{36b},
        def{94d} token,
        NEST1 LAYER new DECSETY1 DECSETY INK module series
          with DECSETY without DECSETY1{d},
        fed{94d} token,
        where <LAYER> is <new DECSETY2 INKSETY>.
d)  NEST3 module series with DESCETY without DECSETY1{c} :
      NEST3 module prelude with DECSETY without DECSETY1{e},
        NEST3 module postlude{f} option.
e)  NEST3 module prelude with DECSETY1 without DECSETY2{d,e} :
      strong void NEST3 unit{32d}, go on{94f} token,
        NEST3 module prelude with DECSETY1 without DECSETY2{e} ;
      where<DECSETY1 without DECSETY2> is
          <DECSETY3 DECSETY4 without DECSETY5 DECSETY6>,
        NEST3 declaration with DECSETY3 without DECSETY5{41e},
        go on{94f} token,
        NEST3 module prelude with DECSETY4 without DECSETY6{e} ;
      where <DECSETY1 without DECSETY2> is <EMPTY without EMPTY>,
        strong void NEST3 unit{32d} ;
      NEST3 declaration with DECSETY1 without DECSETY2{41e}.
f)  NEST3 module postlude{d} :
      postlude{94d} token, strong void NEST3 series with EMPTY{32b}.
g)* module text :
      NEST module text publishing REVS defining LAYER{c}.

{Examples:
  a)  MODULE A = DEF STRING s; read(s);
                     PUB STRING t = "file"+s, PUB REAL a FED,
               B = ACCESS A DEF PUB FILE f;
                                open(f, t, standin channel)
                                POSTLUDE close(f) FED
  b)  A = DEF STRING s; read(s);
              PUB STRING t = "file"+s, PUB REAL a FED  .
      B = ACCESS A DEF PUB FILE f;
                     open(f, t, standin channel)
                     POSTLUDE close(f) FED
  c)  DEF STRING s; read(s);
          PUB STRING t = "file"+s, PUB REAL a FED  .
      ACCESS A DEF PUB FILE f;
                 open(f, t, standin channel) POSTLUDE close(f) FED
  d)  STRING s; read(s); PUB STRING t = "file"+s, PUB REAL a  .
      PUB FILE f; open(f, t, standin channel) POSTLUDE close(f)
  e)  STRING s; read(s); PUB STRING t = "file"+s, PUBLIC REAL a  .
      PUB FILE f; open(f, t, standin channel)
  f)  POSTLUDE close(f) }
```

{Rule b ensures that a unique 'TAU' is associated with each module-text accessible from any given point in the program. This is used to ensure that an 'invoked TAU' can be identified (7.2.1.a) in the nest of all descendent constructs of any access-clause or module-text which invokes that module-text.

In general, a module-text-publising-REVS-defining-LAYER T makes 'LAYER' visible within itself, and makes the properties revealed by 'REVS' visible wherever T is accessed. 'LAYER' includes both a 'DECSETY' corresponding to its public declarations (e.g. t and a in the first module-text of example c), a 'DECSETY1' corresponding to its hidden declarations (e.g. s in that example) and an 'INK' which links T to its unique associated 'TAU' and signifies in the nest that T is now known to be invoked. 'REVS' always reveals 'DECSETY INKSETY INK' (but not 'DECSETY1'), where 'INKSETY' signifies the invocation of any other modules accessed by T. 'REVS' may also reveal the publications of the other modules accessed by T if their module-calls within T contained a public-token.}

4.9.2. Semantics

a) A "module" is a scene {2.1.1.1.d} composed of a module-text together with an environ {2.1.1.1.c}.

b) 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 {c} of the module-text of D1 is ascribed {4.8.2.a} to the defining-module-indication of D1.

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

d) A module-prelude C in an environ E is elaborated as follows:
. its unit or declaration is elaborated in E;
If another module-prelude D is directly descended from it,
then D is elaborated in E
{; otherwise, the elaboration of C is completed}.

{{The declarations in a module-prelude must contain public-symbols if they are to be visible when the module is accessed.}}

4.1.1.

```
A) COMMON :: ... ; module.

e) NEST declaration with DECSETY without DECSETY1{49e} :
     where <DECSETY without DECSETY1> is <EMPTY without DECS1>,
       NEST COMMON declaration of DECS1{42a,43a,44a,e,45a,49a,-} ;
     where <DECSETY without DECSETY1> is <DECS without EMPTY>,
       public{94d} token,
       NEST COMMON declaration of DECS{42a,43a,44a,e,45a,49a,-} ;
     where <DECSETY without DECSETY1> is
         <DECSETY without DECS1 DECSETY2>,
       NEST COMMON declaration of DECS1{42a,43a,44a,e,45a,49a,-},
       and also{94f} token,
       NEST declaration with DECSETY without DECSETY2{e} ;
     where <DECSETY without DECSETY1> is
         <DECS DECSETY3 without DECSETY1>,
       public{94d} token,
       NEST COMMON declaration of DECS{42a,43a,44a,e,45a,49a,-},
       and also{94f} token,
       NEST declaration with DECSETY3 without DECSETY1{e}.
```

{{Modules may be invoked by means of access-clauses.}}

```
1.2.2.
  A)  ENCLOSED :: ... ; access.


  3.6. Access clauses


  3.6.1. Syntax

  a)  SOID NEST access clause{5D,551a,A341h,A349a} :
        NEST revelation publishing EMPTY defining LAYER{b},
          SOID NEST LAYER ENCLOSED clause{a,31a,33a,c,d,e,34a,35a,-}.
  b)  NEST revelation publishing REVSETY
            defining new DECSETY INKSETY{a,49c} :
        access{94d} token,
          NEST joined module call publishing REVSETY revealing REVS{c},
          where DECSETY INKS revealed by REVS{e,f}
            and NEST filters INKSETY out of INKS{h}.
  c)  NEST joined module call publishing REVSETY revealing REVS{b,c} :
        NEST module call publishing REVSETY revealing REVS{d,-} ;
        where <REVSETY> is <REVSETY1 REVSETY2>
            and <REVS> is <REVS1 REVS2>,
          NEST module call publishing REVSETY1 revealing REVS1{d,-},
          and also{94f} token,
          NEST joined module call publishing REVSETY2 revealing REVS2{c}.
  d)  NEST module call publishing REVSETY revealing REVS{c} :
        where <REVSETY> is <EMPTY>,
          module REVS NEST applied module indication with TAB{48b} ;
        where <REVSETY> is <REVS>,
          public{94d} token,
          module REVS NEST applied module indication with TAB{48b}.
  e)  WHETHER DECSETY1 DECSETY2 INKS1 INKSETY2 revealed by
            TAU reveals DECSETY1 INKS1 REVSETY3
            TAU reveals DECSETY1 INKS1 REVSETY4{b,e,f} :
        WHETHER DECSETY1 DECSETY2 INKS1 INKSETY2 revealed by
            TAU reveals DECSETY1 INKS1 REVSETY3 REVSETY4{e,f}.
  f)  WHETHER DECSETY1 DECSETY2 INKS1 INKSETY2 revealed by
            TAU reveals DECSETY1 INKS1 REVSETY2{b,e,f} :
        WHETHER DECSETY2 INKSETY2 revealed by REVSETY2
            and DECSETY1 independent DECSETY2{71a,b,c}.
  g)  WHETHER EMPTY revealed by EMPTY{e,f} : WHETHER true.
  h)  WHETHER NEST filters INKSETY1 out of INKSETY INK{b} :
        unless INK identified in NEST{72a},
          WHETHER <INKSETY1> is <INKSETY2 INK>
            and NEST INK filters INKSETY2 out of INKSETY{h,i} ;
        where INK identified in NEST{72a},
          WHETHER NEST filters INKSETY1 out of INKSETY{h,i}.
  i)  WHETHER NEST filters EMPTY out of EMPTY{h} : WHETHER true.

  {Examples:
    a)  ACCESS A, B (get(f, a); print(a))
    b)  ACCESS A, B
    c)  A, B
    d)  A  .  PUB B }
```

{In rule b, the 'invoked TAU's enveloped by 'INKS' represent those modules which might need to be invoked at any module-call whose applied-module-indication identified a particular defining-module-indication, whereas those enveloped by 'INKSETY' represent only those which need invocation in the particular context, the remainder having already been elaborated, as can be determined statically from the 'NEST'. The presence of 'INKSETY' in the nest of all descendent constructs of the access-clause ensures that all modules now invoked will never be invoked again within those descendents.

Rule f ensures the independence of declarations revealed by one revelation; thus

```
    MODULE A = DEF PUB REAL x FED, B = DEF PUB REAL x FED;
    ACCESS A, B (x)
```

is not produced. However, rule e allows a given declaration to be revealed by two public accesses of the same module, as in

```
    MODULE A = DEF PUB REAL x FED;
    MODULE B = ACCESS PUB A DEF REAL y FED,
           C = ACCESS PUB A DEF REAL z FED;
    ACCESS B, C (x+y+z)
```

in which the module-definitions for both B and C reveal x, by virtue of the PUB A in their constituent revelations.}

{{Note that a particular-program may now consist of a joined-label-definition followed by an access-clause. The defining-module- indications identified thereby would be in the library-prelude or the user-prelude.}}

3.6.2. Semantics

a) A SOID-NEST-access-clause N, in an environ E, is elaborated as follows:
If there exists a "first uninvoked" {b} module M of the revelation R of N in E, with respect to 'NEST',
then
  . let M be composed from a module-text-defining-new-PROPSETY-INK T {together with a necessary environ};
  . M is invoked {c} in E, giving rise to a new environ E4 {inside whose locale 'INK' accesses the result of invoking M};
  . let Y be the yield {a} in E4 of a SOID-NEST-INK-access-clause akin to N {, in which M will be known to be already invoked};
  . {M is revoked, i.e.} the series of the constituent postlude, if any, of T is elaborated in E4;
  . the yield of N in E is Y;
  . it is required that Y be not newer in scope than E;
otherwise,
  . let E2 be the environ established around and beside E according to R {the locale of E2 corresponds to the publicized properties of the modules accessed by R};
  . E2 is "furnished" {d} with {the values publicized by the constituent module-calls of} R in E;
  . the yield of N in E is the yield of the ENCLOSED-clause of N in E2;

b) The "first uninvoked" module of a revelation R in an environ E is determined, with respect to some 'NEST', as follows:
If there exists some constituent module-call-revealing-REVSETY-TAU-reveals-PROPSETY-INK C of R such that the predicate 'unless INK identified in NEST' holds, and which is the textually first such module-call,
then
  . let the yield of the applied-module-indication of C in E be a {not yet invoked} module M composed of a module-text T and an environ E1 {necessary (7.2.2.c) for T};
  If T has a revelation S,
     and if there exists a first uninvoked module M1 of S in E1 with respect to 'NEST',
  then M1 is the first uninvoked module of R;
  otherwise, M is the first uninvoked module of R;
otherwise, there is no first uninvoked module of R.

{Observe that the choice of C from among the module-calls of R depends only on 'NEST' and not on E. It follows, therefore, that the choice can always be made at compile time. E is only required in order to obtain the correct necessary environ for M.}

c) A module composed of a module-text-defining-new-PROPSETY-INK T and an environ E1 {necessary for T} is invoked in an environ E as follows:
If T has a{n already invoked} revelation S,
then
 . let E2 be the environ established around E1, beside E, according to S;
 . the locale of E2 is "furnished" {d} with {the values publicized by the descendent module-calls of} S in E;
otherwise, let E2 be E1;
. let E3 be the environ established around E2 and, if E is a "module locating environ" {see below}, then beside E and otherwise upon E, according to T {the locale of E3 corresponds to all the properties (publicized or not) declared in T};
. 'INK' is made to access the module composed of T and E3 inside the locale of E3 {so that, within T, T itself will be seen to be already invoked};
. the constituent module-prelude of T is elaborated in E3;
. let E4 be the environ, known as a "module locating environ", established around E, beside E3, according to some NOTION-defining-new-INK;
. 'INK' is made to access the module composed of T and E3 inside the locale of E4;
. the invoking of M is said to "give rise" to the environ E4.

{Observe that all the environs created during the invocation of the uninvoked modules (b) of the revelation of an access-clause N have the same scope, which is newer than that of the environ in which N is being elaborated but older than that of any environ created during the elaboration of the ENCLOSED-clause of N.}

d) A locale L is "furnished" with a revelation R in an environ E as follows:
For each descendent module-REVS-applied-module-indication of R,
 For each 'TAU reveals PROPS' enveloped {1.1.4.1.c} by 'REVS',
  . let the module "accessed" {e} by 'invoked TAU' inside E {it will be found in some module locating environ (c)} be a{n already invoked} module composed of a module-text T and an environ E3 {in which its module-prelude was formerly elaborated};
  For each value or scene accessed inside the locale of E3 by some 'PROP',
   If 'PROPS' envelops that 'PROP' {'PROP' is to be publicized},
   then 'PROP' is made to access that value or scene (if it does not so access it already) inside L also.

e) The value or scene "accessed" by a 'PROP' inside an environ E, composed of a locale L and an environ E1, is the value or scene accessed by 'PROP' inside L {2.1.2.c}, if L corresponds to a 'PROPSETY' enveloping {1.1.4.1.c} that 'PROP', and, otherwise, the value or scene accessed by 'PROP' inside E1.

{{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
access symbol{36b}	ACCESS
def symbol{49c}	DEF
fed symbol{49c}	FED
public symbol{36d,41e}	PUB
postlude symbol{49f}	POSTLUDE

{{Moreover, two more new symbols are yet to be invented for use in separate compilation:}}

formal nest symbol{56b}	NEST
egg symbol{A6a,c}	EGG

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

{{Identification}}

```
7.2.1.c+2  # , =>
             or <QUALITY1> is <module REVS> or <QUALITY1> is <invoked>, #
```

{{The proper identification of indicators declared via module-calls is ensured as follows:}}

```
3.0.1.
  f)* NEST range : ... ;
        NEST module text publishing REVS defining LAYER{49c,-} ;
        NEST LAYER1 LAYER2 module series
            with DECSETY without DECSETY1{49d} ;
        SOID NEST access clause{36a}.
```

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 one or {possibly} more applied-module-indications I3 directly descended from NEST-module-calls-revealing-REVS where 'REVS' envelops 'QUALITY TAX'. I1 is then said to "identify" that I2 or each of those I3.

{{This is sufficient to ensure, in conjunction with 7.2.2.c, the proper scope for routines containing access-clauses.}}

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

{{The 'PROPSETY' to which a locale corresponds may now include an 'INKSETY'.}}

2.1.1.1.b+1,+2,+4 # LABSETY => LABSETY INKSETY #

{{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 and a revelation must be establishing-clauses.}}

3.2.1.
```
  i)* establishing clause : ... ;
        NEST module text publishing REVS defining LAYER{49c,-} ;
        NEST revelation publishing REVSETY defining LAYER{36a,-}.
```

Part II - Separate Compilation

{{Separate compilation is performed by dividing a program into packets. Some packets contain formal-holes, indicated by the nest-symbol, into which actual-holes, contained in other packets and indicated by the egg-symbol, may be stuffed.}}

5.1.
```
  A)  UNIT :: ... ; formal hole ; virtual hole.
```

5.6. Holes

5.6.1. Syntax

A) LANGUAGE :: algol sixty eight.
   Extra hypernotions {e.g. "fortran"} may be added to the above metaproduction rule.

B) ALGOL68 :: algol sixty eight.

```
  a)  strong MOID NEST virtual hole{5A} :
        virtual nest symbol, strong MOID NEST closed clause{31a}.
  b)  strong MOID NEST formal hole{5A} :
        formal nest{94d} token, MOID LANGUAGE indication{e,f,-},
          hole indication{d}.
  c)  MOID NEST actual hole{A6a} :
        strong MOID NEST ENCLOSED clause{31a,33a,c,34a,35a,36a,-}.
  d)  hole indication{b} :
        character denotation{814a} ; row of character denotation{83a}.
  e)  MOID ALGOL68 indication{b} : EMPTY.
```
f) Additional hyper-rules, for hypernotions of the form "MOID LANGUAGE indication" are to be added for each extra terminal metaproduction of "LANGUAGE", each containing just one alternative, which is to be a distinct 'bold TAG token'.

{These MOID-LANGUAGE-indications may have severely restricted 'MOID's. For example, the following has been suggested:
```
    FORT fortran indication :
      bold letter f letter o letter r letter t
          letter r letter a letter n token.
where
    LANGUAGE :: ... ; fortran.
    FORT :: procedure with PERFORMERS yielding FOID ;
      procedure yielding FOID.
    PERFORMERS :: PERFORMER ; PERFORMERS PERFORMER.
    PERFORMER :: FODE parameter.
    FODE :: FAIN ; FORT ; reference to FAIN ; ROWS of FAIN.
    FAIN :: real ; long real ; integral ; COMPLEX ; boolean.
    COMPLEX :: structured with real field letter r letter e
          real field letter i letter m mode.
    FOID :: FAIN ; void.
```

Although FORTRAN is now a fortran-indication, it may still be used, if desired, as an operator or as a mode-indication.}

```
{Examples:
  b)  NEST "abc"
  c)  ACCESS A,B (x:=1; y:=2; print(x+y))
  d)  "a"  .  "abc" }
```

{Since no representation is provided for the virtual-nest-symbol, the user is unable to construct virtual-holes for himself, but a mechanism is provided (10.6.2.a) for constructing them out of formal- and actual-holes.}

{The yield of a virtual-hole is that if its closed-clause, by way of pre-elaboration (2.1.4.1.c). No semantics for formal- or actual-holes is provided since their elaboration is never called for.}

{{There are some implementation difficulties in determining the scope of a routine whose routine-text contains a formal-hole, since there is no knowing what indicators may be applied in the actual-hole eventually supplied.}}

7.2.2.c is modified as follows:

```
  ...
    If C contains any QUALITY-applied-indicator-with TAX
      •  ...
      •  ...
      •  ...
    or if C contains a virtual-hole,
    then E is E1;
    ...
```

{{Thus a formal-hole F behaves for scope purposes as if the actual-hole stuffed in its place contained identifiers identifying defining occurrences in every range containing F.}}

{{The packets to be submitted to the compiler for separate compilation may be module-declarations or actual-holes (or particular-programs) and, if they are to be stuffed into formal-holes (rather than into the standard environment), they are introduced by egg-symbols.}}

10.6. Packets

10.6.1. Syntax

```
a)  MOID NEST new MODSETY ALGOL68 stuffing packet{A7a} :
      egg{94d} token, hole indication{56d}, is defined as{94d} token,
        MOID NEST new MODSETY actual hole{56c}.
```

b) Additional hyper-rules, for hypernotions of the form "MOID NEST new MODSETY LANGUAGE stuffing packet" are to be added for each extra {5.6.1.A} terminal metaproduction of "LANGUAGE". A mechanism must be defined {presumably with the aid of the Report defining that other language} whereby all such LANGUAGE-stuffing-packets may be transformed into ALGOL68-stuffing-packets {with the same meaning}.

```
c)  NEST new MODSETY1 MODS definition module packet of MODS{A7a} :
      egg{94d} token, hole indication{56d}, is defined as{94d} token,
      NEST new MODSETY1 MODS module declaration of MODS{49a},
      where MODS absent from NEST{e}.
d)  new LAYER1 new DECS MODSETY1 MODS STOP
        prelude packet of MODS{A7a} :
     new LAYER1 new DECS MODSETY1 MODS STOP
        module declaration of MODS{49a},
      where MODS absent from  new LAYER1{e}.
e)  WHETHER MODSETY MOD absent from NEST{c,d} :
     WHETHER MODSETY absent from NEST{e,f}
        and MOD independent PROPSETY{71a,b,c},
      where PROPSETY collected properties from NEST{g,h}.
f)  WHETHER EMPTY absent from NEST{e} : WHETHER true.
g)  WHETHER PROPSETY1 PROPSETY2 collected properties from
        NEST new PROPSETY2{e,g} :
     WHETHER PROPSETY1 collected properties from NEST{g,h}.
h)  WHETHER EMPTY collected properties from new EMPTY{e,g} :
     WHETHER true.
i)* NEST new PROPSETY packet :
     MOID NEST new PROPSETY LANGUAGE stuffing packet{a,b} ;
     NEST new PROPSETY definition module packet of MODS{c} ;
     NEST new PROPSETY particular program{A1g} ;
     NEST new PROPSETY prelude packet of MODS{d}.
j)* letter symbol : LETTER symbol{94a}.
k)* digit symbol : DIGIT symbol{94b}.
```

{Examples:

```
  a)  EGG "abc" = ACCESS A,B (x:=1; y:=2; print(x+y))
  c)  EGG "abc" = MODULE A = DEF PUB REAL x FED
  d)  MODULE B = DEF PUB REAL y FED
```

The three examples above would form a compatible collection of packets (10.6.2.a) when taken in conjunction with the particular-program

```
     BEGIN NEST "abc" END }
```

{In rule a above, 'MODSETY' envelops the 'MOD's defined by all the definition-module-packets that are being stuffed along with the stuffing-packet. In rules c and d, 'MODSETY1' need only envelop the 'MOD's for those modules actually accessed from within that packet. The semantics below are only defined if, for a collection of packets being stuffed together, all the 'MOD's enveloped by the various 'MODSETY1's are enveloped by 'MODSETY'.}

10.6.2. Semantics

{Packets are the units of separate compilation. It is necessary to define the meaning of a collection of packets. This is done by transforming the collection into an equivalent particular-program. It is, of course, necessary for the packets of the collection to be compatible with each other. Just one of the packets must be a particular-program.}

a) The meaning of a particular-program P, in the context of a collection of other associated packets {not particular-programs} T, is determined as follows:

. The user-prelude-with-MODSETY UP of the user-task UT from which P is descended {1.1.1.e and 10.1.1.f} must be composed as follows:

For each new-LAYER1-new-DECS-MODSETY1-STOP-prelude-packet M, if any, in T,

. UP contains a constituent new-LAYER1-new-DECS-MODSETY-STOP-module-declaration akin to the module-declaration of M; {'MODSETY' must envelop all the 'MOD's enveloped by all such 'MODSETY1's, and no others, for the user-prelude of U to be syntactically correct;}

. UP contains no other constituent COMMON- declarations, and its only

constituent unit is composed of a skip {5.5.2.1.a};
If T contains any LANGUAGE-stuffing-packets, where 'LANGUAGE' is not 'ALGOL68',
then those packets are transformed {10.6.1.b} into ALGOL68-stuffing-packets {with the same meanings};
While there remain any formal-holes in UT,
. let H be one such MOID-NEST-formal-hole and let I be its hole-indication;
. if I is akin to any such I previously considered, then the meaning of P is not defined;
. H is replaced {in UT} by a MOID-NEST-virtual-hole whose constituent NEST-serial-clause S is composed as follows:
For each NEST-new-MODSETY1-definition-module-packet M, if any, in T whose hole-indication "matches" {b} I,
. S contains a constituent NEST-new-MODSETY-module-declaration akin to the module-declaration of M; {'MODSETY' must envelop all the 'MOD'S enveloped by all such 'MODSETY1's, and no ohers, for S to be syntactically correct;}
. S contains no other constituent COMMON-declarations, and its only constituent unit is composed of the constituent ENCLOSED-clause of the {only} MOID-NEST-new-MODSETY-ALGOL68-stuffing-packet in T whose hole-indication matches I;
If there remain any packets in T that have not been incorporated into U,
then the meaning of P is not defined;
otherwise, {UT does not contain any formal-holes, and therefore} the meaning of P is as defined elsewhere {1.1.1.e} by the semantics of the Report.

b) If the {textually} first constituent string-item of a hole-indication I is composed of some letter-symbol and each other constituent string-item, if any, is composed of some letter-symbol or some digit-symbol, the I "matches" any other hole-indication to which it is akin {; otherwise, its matching with other hole-indications (whethr akin or not) is not defined here, but may be defined by local conventions of the implementation to suit the peculiarities of the local operating environment}.

{{The standard environment is enlarged by the inclusion of a user-prelude for each particular-program, into which the user may stuff his own prelude-packets.}}

10.1.1.

A) EXTERNAL :: ... ; user.

```
f)  NEST1 user task{d} :
        NEST2 particular prelude with DECS{c},
          NEST2 user prelude with MODSETY{c},
          NEST2 particular program{g} PACK, go on{94f} token,
          NEST2 particular postlude{i},
          where <NEST2> is <NEST1 new DECS MODSETY STOP>.
```

10.1.2
f) Except where explicitly stated otherwise {10.6.2.a}, each constituent user-prelude of all program-texts is EMPTY.

Part III - Compilation Systems

{{Although the Report defines the meaning of a particular-program (and, with the addition of the new section 10.6, of a collection of compatible packets) without reference to the process of compilation (except pragmatically in 2.2.2.c), a proposal for separate compilation will not be of practical use unless the majority of implementations observe at least some degree of consistency in their compilation systems.}}

10.7. Compilation systems

An implementation of ALGOL 68 {2.2.2.c} in which packets of a {compatible} collection {10.6.2} are compiled into a collection of object-modules should conform to the provisions of this section.

10.7.1. Syntax

A)* LAYERS :: LAYER ; LAYERS LAYER.

```
a)  compilation input :
      MOID NEST new MODSETY LANGUAGE stuffing packet{A6a,b},
        MOID NEST hole interface{d},
        joined module interface with MODSETY{b,c} ;
      NEST new MODSETY1 MODS definition module packet of MODS{A6c},
        MOID NEST hole interface{d},
        joined module interface with MODSETY1{b,c},
        module interface with MODS{d} option ;
      new LAYER1 new DECS MODSETY STOP particular program{A1g},
        {void new LAYER1 new DECS STOP hole interface,}
        unless <DECS> contains <module>,
        joined module interface with MODSETY{b,c} ;
      new LAYER1 new DECS MODSETY1 MODS STOP
          prelude packet of MODS{A6d},
        {void new LAYER1 new DECS STOP hole interface,}
        unless <DECS> contains <module>,
        joined module interface with MODSETY1{b,c},
        module interface with MODS{d} option.
b)  joined module interface with MODS MODSETY{a,b} :
      module interface with MODS{d},
        joined module interface with MODSETY{b,c}.
c)  joined module interface with EMPTY{a,b} : EMPTY.
```

d) Hyper-rules are to be added for the hypernotions "MOID NEST hole interface", "module interface with MODS" and "MOID NEST object module" {the first two to be} such that, from the terminal production of each MOID-NEST-hole-interface (each module-interface-with-MODS), a 'MOID1 NEST1' equivalent {2.1.1.2.a} to 'MOID NEST' (a 'MODS1' equivalent to 'MODS') can be reconstructed. {The forms of these hyper-rules are otherwise undefined, and their terminal productions will most probably be in some cryptic notation understood only by the compiler.}

{The inclusion of the hypernotions "void new LAYER1 new DECS STOP hole interface" within pragmatic remarks in rule a is intended to signify that this information (which describes the standard environment) must clearly be available to the compiler, but that it may well not be provided in the form of an explicit hole-interface.}

10.7.2. Semantics

a) A compilation-input C may be compiled by a compiler. The output from the compiler is determined as follows:
Case A: the packet of C is a MOID-NEST1-ALGOL68-stuffing-packet:
- the compiler-output is a MOID-NEST1-object-module;

Case B: the packet of C is a NEST1-particular-program:
- the comiler output is a void-NEST1-object-module;

Case C: the packet of C is a NEST1-definition-module-packet-of-MODS or a NEST1-prelude-packet-of-MODS D:
- the compiler output consists of
  (i) a void-NEST1-object-module, and
  (ii) if the module-interface-with-MODS-option of D is EMPTY, a module-interface-with-MODS {; otherwise, the constituent module-interface-with-MODS of D is said to be an "imposed interface" (obtained from the previous compilation of a similar packet) and the compiler must fail if the imposed interface is no longer "consistent" with the packet};

{Case D: the packet of C is a LANGUAGE-stuffing-packet where 'LANGUAGE' is not 'ALGOL68':
- the compilation process is not defined by this Report;}

For each MOID-NEST-LAYERS-formal-hole contained in the NEST-packet of C,
- the compiler output includes, additionally, a MOID-NEST-LAYERS-hole-interface.

b) The module-interfaces and hole-interfaces output by the compiler may subsequently be used, together with appropriate packets, as compiler-inputs. If a collection of packets, including a particular-program P whose meaning is defined {10.6.2.a} in the context of that collection, is compiled so as to produce a corresponding set of object-modules, then the meaning of those object-modules is the same as the meaning of P.

{A complete system may include a compiler, a loader, and a means to maintain a library of packets, hole-interfaces, module-interfaces and object-modules (the means might be an operating system, a utility program written for the purpose, or a filing cabinet plus a little girl). The assemblage of the various objects required for a compilation-input and the disposal of the various compiler outputs may involve the user in writing control cards, or pragmats, or other forms of command, and in providing libraries of such objects to be scanned. Neither the detailed contents of such a system nor the specific forms of such commands are defined in this Report.

If a packet P is modified and re-compiled, the system should ensure that the revised collection of object-modules cannot be used until all packets dependent upon P have been re-compiled. It is suggested that all the outputs produced by a given compilation be given a unique serial number from a monotonically increasing set (the date and time, for example) and that object modules be aware of the serial numbers of other compilations upon which their validity depends. However, where the compiler detects that a hole- or module-interface is unchanged from a previous compilation of the same packet, or if a module-interface is imposed on a compilation and the compiler is able to produce an object-module "consistent" with that module-interface, then the old serial number may be retained. The definition of "consistent" should be as liberal as possible. For example, it should be possible for the compiler to compile a packet consistent with the object-module produced by a previous compilation of that packet even if the indicators published by the packet are now declared in a different order or if declarations for additional indicators have been added.}

Implementation methods for Modules and Separate compilation.

This implementation description does not contain language definition. It presents various ways in which the above features can be implemented. No implementer should feel committed to do things as described here, though he may well profit from the thought that has gone into these methods. The same language facilities may well be implementable in other ways. Two mechanisms are described. One is a mechanism for implementing separate compilation, and the other a mechanism for implementing definition modules.

The notation "MR" will be used to refer to the Revised Report as extended by the Formal Definition above.

1 Separate compilation.

The separate compilation methods for the features defined above hinge on the idea of a "compilation data base". This data base contains information about the various separately compiled parts of a program, and is used to enable static mode checking to be done across compilations and to enable efficient object code to be generated. The data base contains information grouped into "interfaces". Each interface contains the relevant information from a single separate compilation and is constructed by the compiler in addition to the usual object code. When a program is compiled whose meaning depends on other separately compiled parts, the compiler extracts the relevant interfaces from the data base. The data base itself may be implemented in different ways, depending on the implementation environment. It may, for example, be managed directly by the compiler, by an operating system which demands its own extra control cards, or even by a clerk with a drawer full of paper tapes. If the operating system's file system is divided into subsets for various users with varying access rights, it is probably wise to permit the data base to be spread out throughout the operating systems's files in the same way. Each user then has control of that part of the data base that relates to his own programs, without requiring installation management to set up separate administration procedures for ALGOL 68.

The production rules which follow often contain ampersands ("&") instead of commas. This is to indicate that the various members must be available in some form, but that nothing is said about their textual order, or even whether a textual order exists. The data may legitimately reside in an arbitrarily inscrutable data base management system and be pieced together by the compiler.

1.1 Compilation input.

```
compilation input :
    definition module packet &
       imposed module interface option &
       joined module interface {for definition modules, if any,
          accessed by this one} &
       hole interface option {if we are inside a hole} ;
    particular program &
       joined module interface {for definition modules accessed
          by the particular program} ;
    stuffing packet &
       hole interface &
       joined module interface {for definition modules, if any,
          accessed by the stuffing}.
```

```
source packet :
    definition module packet ;
    particular program ;
    stuffing packet.
```

The programmer writes source-packets.

```
joined module interface :
    set of module interfaces.
```

The phrase "set of" is used in its usual mathematical meaning.

```
imposed module interface :
    module interface.

interface:
    hole interface ;
    module interface.
```

Interfaces are not written by the programmer, but are produced by a compiler when a definition-module-packet, or a source-packet containing a hole, is compiled. Interfaces may later be fed back into subsequent compilations or recompilations to ensure compatibility. A single interface may be used in many different compilation-inputs. The syntax and semantics of interfaces are implementation-dependent, but each interface must contain the modes and indications published by the module or available to the stuffing, as well as the "access algorithms" which enable the compiler to generate correct code for applied-indicators in a separate compilation.

If a definition module is altered and recompiled (perhaps to improve performance or to fix a bug), an interface from a previous compilation of that same definition module may be "imposed" in an attempt to ensure compatibility with the existing object code of other packets accessing that definition module. If the compiler is able to achieve compatibility, it does so; otherwise, it will complain and produce incompatible code and a new interface. Clearly, if less information resides in the interface, it will be easier to make program changes, but the resulting object code may be less efficient.

Module-interfaces may be used in several different ways, depending on practical aspects of the implementation.

(1) Bottom-up coding

If a program is being coded bottom-up, with each module thoroughly debugged before the ones that access it, derived interfaces are convenient. When a definition module is compiled, the compiler will produce a module-interface as well as the usual object code. This module-interface must then be fed back into the compiler when the module's test procedures are compiled, and later, when a program accessing the definition module is compiled. This module-interface will be checked for compatibility with its usage in the accessing program, thus maintaining mode security.

(2) Top-down coding

This method is based on the principle that, when programming, the interface between program components should be defined logically before the components are constructed. The programmer (or perhaps his manager) will therefore start by defining an "interface definition" for a definition module. This interface definition is written as a definition-module-packet with skips or holes in the proper places (assuming the compiler does not propagate the skip-value or hole into the interface). It is compiled, the object code is discarded, and the compiler-produced interface is preserved. The interface is presented to the programmer when he writes the definition module. The interface is imposed on the compiler when it compiles the definition module, and is also provided when it compiles the accessing program. In case of incompatibility, the compiler will complain. The access algorithms and other internal implementation information will be determined for the compiler by the interface.

(3) Program libraries.
Interfaces of definition modules in public libraries must also be provided as part of the library. It is up to the library maintainer whether he wishes to use imposed interfaces to make changes less painful.

If a compiler accepts "multiple separate compilations", that is, if it accepts many compilation-inputs at one go, some mechanism (such as library search) should be provided so that a single copy of each module-interface will suffice for all compilations. The existence (yes, mere existence) of multiple (and therefore independent) copies involves the risk that interfaces may not match when separately-compiled packets are loaded and run together.

1.2 Holes

Holes are useful if a large existing program must be cut into pieces, perhaps because it has grown or because it is transported to an installation whose compiler has less capacity. Unlike definition modules, holes permit a program to retain its original structure when it is cut up.

Furthermore, the compile-time flow of information through a hole is exclusively from the root to the leaves of the complete parse tree. Holes may thus be used to prevent a compiler from taking advantage of any knowledge about the contents of a construct. This may be important if parts of a program are to be changed independently.

The hole mechanism has been called a "top-down" method for separate compilation; this is perhaps a misnomer in that in top-down programming the refinements usually consist of new procedures and modules, and not of further contents for holes in a parse tree.

The object code of a hole contains a call to its stuffing, using operating-system external linkage conventions and using the hole-indicator as an external symbol.

It is not necessary to start a new display level for each nested stuffing, but it may be convenient. If this is not done, some stack mechanisms may have difficulty determining the proper activation record size on procedure entry. If the constituent unit of a routine-text is a hole, it may be wise to compile calls to the procedure using the external-indication directly instead of via a dummy routine.

1.3 Module- and hole-interfaces.

This section describes some possible contents for interfaces. Specific implementations may of course do things differently.

```
module interface :
    unique code & external symbol & hole description option &
          mode table & definition summary.

hole interface :
    unique code & external symbol & hole description option &
```

```
        mode table & set of definition extracts.
```

The unique code may be a possibly compactified version of the module-interface, a hash code computed from it, a time stamp, or some other code unique to the interface. These unique codes can be compared at linkedit or run time to check that object codes run together were indeed compiled to a corresponding interface. Because hash codes computed from different interfaces might possibly be duplicates, some implementers might provide a formal interface-registration utility-program which could perform system-wide or library-wide synchronization to prevent inadvertent (but highly unlikely) duplication of codes. Such a registration utility might even be part of the compiler.

The external-symbol must be sufficient to determine the entry-point at which execution of the stuffing or definition module is to begin.

The hole-description-option specifies into which nested sequence of holes, if any, the packet producing the interface is to be nested. This is necessary to check at compile-time that the necessary environment is indeed available at the accession of a definition module, which may have been compiled in a different nested sequence of holes.

The mode-table contains a full description of every mode required in the definition-summary or set-of-definition-extracts. It may have undergone mode equivalencing to reduce redundancy.

The definition-summary contains information about all definitions published by the definition module or hole. Its structure closely follows the metasyntax of REVS {MR1.2.3}.

```
definition summary{REVS} :
    set of definition groups.

definition group{REV} :
    module identity{TAU} & set of definition extracts{DECSETY INKS}.

definition extract{DEC} :
    mode extract{DEC} ;
    operation extract{DEC} ;
    priority extract{DEC} ;
    identifier extract{DEC} ;
    definition module extract{DEC} ;
    invocation extract{INK}.

mode extract :
    mode marker & mode indication & mode & mdextra.

operation extract :
    operation marker & operator & mode & mdextra.

priority extract :
    priority marker & operator & integer priority & mdextra.

identifier extract :
    identifier marker & identifier & mode & mdextra.

definition module extract{MOD} :
    definition module marker & definition module indication{TAB} &
       definition summary{REVS} & mdextra.

invocation extract{INK} :
    module identity{TAU}.
```

mdextra :
    extra machine-dependent information.

The extracts are sufficient to enable reasonable object code to be generated to access the publications of a definition module without any further information in the mdextra, since a compiler can use a canonical algorithm to determine the access algorithms for the published indicators. Hole-interfaces, however, will likely be far more complicated, and may require extra machine-dependent information to be recorded in the mdextras, such as display-nesting and displacements for the various indicators. Extracts should be kept as nonspecific as is compatible with efficiency, because every datum in the interface makes compatible compilation of a new version of a packet more difficult. The indicators published by definition modules can more easily be forced into a canonical format that depends only upon the DECs than can the indicators from a hole-interface.

If optimization of object-time code is more important than program flexibility, the compiler can place further implementation-dependent information into the mdextras. It may, for example, include the values of known constant indicators, side effect information about procedures, or even a partially-compiled version of the source code for routines it may wish to compile in-line.

2 Implementation of definition modules.

2.1 Notation

The text of a definition module M may begin with a joined-module-call. Each module-call of the joined-module-call will be called a "requirement" of M, and M is said to "require" the corresponding definition modules.

2.2 General strategy.

A definition module can be implemented like a procedure. When it is invoked, it accesses any definition modules it itself requires and allocates an activation record on the stack just as a procedure would, and then executes its prelude. It then returns to its invoker, passing the address of its activation record to the invoker without freeing its local storage. The invoker can find the published indicators within the activation record, and when the time comes to revoke the definition module, the postlude is elaborated. Only afterward is the stack frame for the definition module released. Slight variations on this scheme are possible. For example, if the invoker knows the necessary size, the definition module's activation record can be allocated within that of the invoker. (This optimization is possible with nonrecursive procedures as well.)

Section 2.3 describes the run-time activity necessary for implementation in further detail. Section 2.4 describes how definition modules can be fitted into existing ALGOL 68 parsing techniques.

2.3 Implementation of sharing

There are several methods of implementing sharing, that is, of deciding whether a module-call actually requires a definition module to be executed, or whether it merely accesses a former invocation. It can be done completely at compile-time, it can be done completely at run time, and mixtures of these two methods are also possible. The compile-time methods are simpler, but the run-time methods are more flexible. The run-time methods are recommended during program development, since otherwise, as we shall see, internal changes in a definition module may cause much accessing code to be recompiled even if it is not changed.

Section 2.3.1 presents a possible strategy for implementing definition modules, on the assumption that all sharing decisions are made at compile time. After that, in section 2.3.2, the necessary modifications are described for doing this at run time.

2.3.1 Compile-time sharing

This method is possible if it is known at compile-time whether each module-call involves an actual invocation or merely accesses some previous invocation. This information is available if:

(1) the definition module in question is part of the same compilation-packet as its module-calls(s), and no possibility exists for any unknown accessions from separate compilations, or

(2) the compiler always places all the INKS in the interface-packet of a definition module.

Under these assumptions, when a compiler comes to compile a module-call of a definition module M, it first tests whether the NEST includes an INK from another module-call of the same module. If so, no actual invocation is done, and in the closed-clause or definition module which uses the accession, code is generated to refer to the activation record from the older invocation instead.

If the accession involves an actual invocation, the compiler first checks whether M has any requirements. If so, each of these other definition modules is first accessed. This is a recursive process, involving the entire mechanism of NEST searches, accessing further requirements, etc. Afterward, M is called, with the pointers to the activation records of the requirements as parameters.

The entry-point used for calling M is the beginning of its prelude. The return address is the beginning of the code that may use the publications of the invocation; in the case of a joined-module-call this will be the next module-call on the list, if any.

Upon entry, M first establishes an activation record for its own use. If the size of this activation record is known by the invoker, the invoker can have allocated it as part of its own activation record and can have passed the address of the activation record to M as parameter.

The prelude of M is then elaborated. Within M, and within any procedures within M, local and global variables are obtained via a normal display or static chain mechanism starting from the new activation record.

At the end of the prelude, M returns, without freeing its activation record. If M allocated its own activation record, it passes the activation record pointer back to the invoker. The code which uses the publications of M is then executed. The publications of M can be reached by displacements from the activation record pointer. If the activation record was part of the invoker's activation record, different displacements from the start of the invoker's own activation record can be computed at compile time and used instead.

When its time comes, M is revoked by calling its postlude, if any, providing it with the address of the activation record of M in some way. The postlude is elaborated, and returns, again without freeing the activation record of M.

Back at the invoker, the definition modules invoked as requirements of M are also revoked. When all the definition modules involved in the access-clause have been duly revoked, the activation records can all be freed by reducing the stack pointer.

If labels were to be permitted in postludes (they are not, but an implementer might wish to implement the stop from the standard-postlude in this way), it might be possible for the prelude to go to the postlude directly instead of waiting for an honest revocation. To avoid trouble, an extra return address would have to be provided when the prelude is called to enable the postlude to return properly. This return address would be that normally provided when the postlude is called. It is to prevent this and other worse obscurities that labels cannot be declared in postludes.

A problem with the above method is that it makes the interfaces for separate compilation unduly restrictive. It becomes difficult, for example, to restructure a large library by organizing its internal procedures into different combinations of definition modules, without requiring massive recompilation of all user code. These problems can be obviated with a suitably clever dedicated linkage editor, but the implementer may not have this freedom.

2.3.2 Run-time sharing

If the above method is not suitable, run-time analysis can be performed for making sharing decisions. These methods do not have the execution efficiency of the compile-time methods, but may have other advantages. In the absence of a special ALGOL 68 linkage editor, the run-time mechanisms may indeed be necessary during program development to retain a modicum of flexibility. They are efficient if definition modules are only rarely accessed. This will hold if definition modules are used mainly for establishing the large-scale structure of the program, and procedure calling is used for normal traffic.

Existing accessions are recorded in an in-core data base at run time. Each accession of some definition module M causes an "activator" to be constructed and placed into the data base. This activator is made to point to a linked list of the activators for the definition modules required by M. These other activators are placed on the list one at a time, as their definition modules are accessed. These activators point to further linked lists. The activators are thus linked together into a tree structure which mimics the INKS {MR3.6.1}. The roots of these activator trees are linked according to the syntactic nesting of activations within the program, from the inside outwards, parallel to the static link. We give the links the following names: the linked lists are linked by the 'next' link, and the sublists are pointed out by the 'sub' link.

An "activator" is thus a structure with fields:

- defmod: the definition module, as an entry-point-environment pair,
- actrec: a pointer to an activation record containing the publications of the definition module,
- revoker: the address of the postlude,
- sub: the address of another activator (which starts a sublist), and
- next: the address of another activator (in the same linked list).

A module-text FOO is accessed as follows:

- The accessor makes a new activator FOO1.
- The accessor fills in the entry point-environment pair of the definition module FOO being accessed into 'defmod of FOO1'.

- The accessor fills in the "next" link of F001 to point
  - if the accessor is the first module-call of the requirements of a module-text,
    - to the activator X created by the accessor's own invoker, or
  - if the accessor is a second or subsequent module-call of a joined-module-call,
    - to the activator of the previous module-call of the joined-module-call, or
  - if the accessor is the first module-call of an access-clause C,
    - to the "principal" activator of the smallest access-clause or module-text containing C, or nil if there is none.
    - This other activator can be found by the same sort of addressing formula as is used for ordinary variables; it is as if each new module-call declared some special indicator and the statically most local definition of the special indicator were always used.
- Then the accessor jumps to a service routine. The service routine receives as parameters
  - a reference to the activator F001, and
  - two labels:

    after_prelude:

    pointing to the controlled-clause of the access-clause (or its analogue for the revelation of a definition module),

    after_postlude:

    pointing to the code to be executed after the postlude has been executed. For the first module-call in the joined-module-call of the access-clause, this will be the address of the code to be executed after the access-clause or module-text. For a second or subsequent module-call of a joined-module-call, this will be the address of an indirect jump to the revoker (see below) of the previous module-call; this revoker is the postlude address of the previous module-call.
- The service routine searches the tree of activators rooted at F001 with branches 'next' and 'sub' to determine whether there is already another activation of the definition module FOO in the tree.
  - If so,
    - the 'revoker of F001' (which contains the address of the postlude) is set to after_postlude (since no actual invocation is done, no actual revocation will be done either).
    - the 'actrec of F001' (the activation record pointer) is filled in with the activation record pointer of the (other) invocation, if any, and otherwise further elaboration is undefined (in this case the other activation record is not yet complete).
  - If not, the accession is actually an invocation, and
    - the object code for the module-prelude of FOO is called, giving it the activator F001 as parameter.
    - FOO receives control, sets up an activation record of its own, and accesses its requirements in order (this will have the effect that the activators of these requirements come to be a linked list linked by the 'next' link and pointed to by 'sub of F001').
    - When FOO's requirements have been met, FOO makes a copy F002 of the activator F001, and sets the next-pointer of the copy F002 to point to the principal activator of the smallest access-clause or module-text containing FOO (or nil if there is none). This copied activator F002 is termed the "principal" activator of FOO, and is used in its prelude's and postlude's own private module-calls. F002 is necessary because the prelude and postlude are in a different NEST from the invoker.
    - When elaboration reaches the end of the prelude, FOO
      - fills the address of the postlude into 'revoker of F001',

  - fills the address of its activation record into F001 and F002, and
  - goes back to the invoker using the after_prelude address without freeing any activation records.
- If the accessor was a requirement of a definition module,
  - the accessor sets 'next of F001' to point to the list of activators of previous requirements of F001 (formerly pointed to by sub of X), and sub of X is updated to point to F001 (this places F001 on the sublist of activators of requirements of X),
- but otherwise, if the accessor was the last module-call of the joined-module-call of an access-clause, F001 is termed the "principal" activator of that access-clause.
- When the definition module is revoked by the accessor, the accessor goes to the routine pointed to by the postlude address of the activator F001. This turns out to be the address of the postlude if the definition module was actually invoked; it is the after_postlude address otherwise.
  Before it finally returns, the postlude revokes the definition modules that FOO accessed.
- When the elaboration of an access-clause is complete, the run-time stack can be cut back to its size before the elaboration of the access-clause started (except that the yield of the clause must be preserved). This frees the activation records and activators of any newly-invoked definition modules without damaging the activation records found via sharing.

Activation records are not freed when elaboration of a module-postlude is complete, even if that definition module invoked other definition modules. They are freed only when some access-clause is complete. In this way the scopes of all activation records created by a single joined-module-call can be the same.

Notice that a jump which jumps out of an invocation will free the activation record by simply popping the stack without executing the postlude. This is consistent with the behaviour of jumps elsewhere.

A "redundant" activator is one which did not cause a new invocation, but simply found an old activation record. If the chain of activators becomes too long, it can be shortened by linking around redundant activators instead of through them.

If any other active activator of a definition module is statically known at the point of activation, that activator can be used instead of repeating any accession overhead.

2.4 Outline of parsing algorithm.

2.4.1 Description

The following processes must be present in some form in an ALGOL 63 compiler.

-1- Distinguishing mode, operation, and priority declarations and determining the ranges in which they hold sway, and building up a definition dictionary containing this information.

-2- Determining whether each applied bold-TAG-symbol is an applied mode, operation, and/or priority indication.

-3- Distinguishing all declarations.

-4- Either from the information from -1- or -3-, constructing a mode table.

-5- Mode equivalencing
-6- Identifying the defining occurrences for all applied indicators.

These processes need not be distinct. Some can be combined easily; others can be combined only if one requires declaration before use. Processes -2- and -3- are often carried out concurrently with context-free parsing. It is at this time that the definitive definition dictionary can be built. It resembles the earlier definition dictionary, but identifier definitions are included as well.

Definition modules are included in this process as follows:

-1- Definition module definitions and accessions are distinguished and entered into the definition dictionary too during process -1-. To each definition module declaration entry, the compiler must attach the set of definitions the definition module itself publishes and the module-indications it publicly accesses. To save space at compile-time, this may be combined with the set of definitions available within the definition module's own range, but a bit must be added to indicate whether each definition or module-call is public. Identifier declarations are not collected, since it is necessary to distinguish mode indications from operators in order to distinguish their declarations.

-2- In process -2-, the applied indications may now turn out to be module indications. Upon range entry, module-calls are identified. When an applied-module-indication has been identified, extra definition entries are added to the definition dictionary for the new range, one for each published definition in the accessed definition module. These extra definition dictionary entries refer to the module-call they arise from. The extra mode, operation, priority, and definition module definitions are thus made available for identification during processing of the range. This second phase is probably the proper moment to perform a library search through the compilation data base for modules which are accessed but not declared by the programmer.

-4- The preliminary mode table can be built only when module-indications have been identified. It must therefore use information from process -3- instead of -1-.

-5- Mode equivalencing occurs as usual.

-6- Coercion and identification occur as usual, too, except that the extra NEST entries created by accessions must also be processed.

2.4.2 Example

Consider the following example:

```
BEGIN #c1# LOC INT b;
   MODULE B = DEF #c2# PUB MODE A = REAL FED;
   BEGIN #c3#
      BEGIN #c4#
      b := 2;
      ACCESS #c5# B
         ( #c6# b := 2; A b; SKIP)
      END;
      MODULE B =
         DEF #c7#
            PUB OP A = (#c8# INT i)VOID: SKIP
         FED;
      SKIP
   END
END
```

In phase -1-, the corrals are identified by the occurrence of BEGIN-END, DEF-FED, and (-) brackets and by ACCESS (a corral is a bracket-pair which might turn out to be a range). Several declarations are detected:

```
    d1. MODULE B in corral c1
    d2. MODE A in corral c2 (published by d1)
    d3. ACCESS B in corral c5
    d4. MODULE B in corral c3
    d5. OP A in corral c7 (published by d4)
```

The identifier declarations have not yet been detected because of uncertainty whether bold words are modes or operators. The next scan over the program now has enough information to identify bold words. At each corral entry, it examines the above table to determine which bold words are defined there.

```
    corral c1:
       MODULE B (which will publish MODE A when accessed)
    corral c2 within c1:
       MODE A
    corral c3 within c1:
       MODULE B (redefining B)(which will publish OP A when accessed)
    corral c4 within c3:
       nothing new
    corral c5 within c4:
       OP A (from ACCESS B)
    corral c6 within c5:
       nothing new
    corral c7 within c3:
       OP A
    corral c8 within c7:
       nothing new
```

Because it is now known which operators and modes are declared where, process -3- can now determine which identifiers are declared for later processes to use:

```
    corral c1 declares LOC INT b.
    corral c6 does not declare A b, because A is an operator there.
    corral c7 declares INT i.
```

Process -3- can still be performed concurrently with process -2-.

The rest of identification and coercion can proceed as usual.

2.5 Avoiding loading of procedures.

If a definition module is used as a library, it may be necessary to avoid loading object code for routines that are not used by the user. Although mechanisms for doing this are inherently implementation-dependent, most loaders have library search facilities for loading only those separately-compiled object files that have been referred to (some loaders can even delete unreferenced fragments of code within a single object file). On such a linking loader, we can use the following mechanism. The body of the routine of a declaration can be a hole:

```
        PROC p = (REAL a, b) REAL: HOLE "foo"
```

It is possible to record this external name "foo" in the interface. An external reference need be present in object code only

- if the procedure is called, or
- if a routine-value is actually required (perhaps to assign it to a procedure variable).

The library search of the linking loader can then be used to ensure that the object code for the procedure, which is compiled separately, is loaded only if needed. It is possible to avoid using holes for this if the compiler is willing to take over program library management completely instead of just producing object code files to be placed into a library by an independent utility. Of course, if the operating system has a half-decent linking loader (most do not), or if the ALGOL 68 implementer provides his own, the above techniques should be unnecessary.

2.6 A use for the escape character.

If it is desired to perform many separate compilations with many different compiler-inputs in one input file using a single run of the compiler, control cards may be needed to separate packets in a way that is independent of syntax errors within the packets. It should be noticed that the standard hardware representation does not enable an ALGOL 68 program line to begin with a single apostrophe (except in comments or pragmats). This may be a natural choice as control-card indicator for some implementations. {Why do we still speak of control cards in the telecommunications age?}

2.7 A new view of the standard prelude.

The thought might be entertained to implement the particular-program as the stuffing of some hole in the standard-prelude. This would be unwise on some implementations, since it would mean that all ALGOL 68 programs would get the same external entry-point name. It may be better to implement the standard-prelude as a collection of definition modules implicitly accessed by all other source-packets. Of course, some kludge will then be necessary for the stop in the standard-prelude. If the standard-prelude should actually be written in ALGOL 68, some mechanism will also be necessary to suppress the implicit accession of the standard prelude when it itself is being compiled.

2.8 A tricky implementation method for strict stack machines.

A "strict stack machine" is a machine whose hardware strictly enforces a procedure-stack memory hierarchy of the style of ALGOL 60. Strict stack machines are difficult to use with unusual control structures because they impose the wrong structure on the program, but definition modules can still be squeezed in.

A definition module can be viewed as a procedure M which accepts as arguments
- an activator A, and
- a procedure P.

It checks whether to make a new invocation, and
- if so, makes a copy of the activator, elaborates the prelude, and fills in appropriate activators, as usual,
- and otherwise, digs up the old invocation.

It then calls P, giving it as parameters
- each published indicator, and
- a procedure Q, which will elaborate the postlude when called.

When P returns, M immediately returns.

An access-clause sets up an activator, and then calls the definition module, giving it as parameters:
- the activator A,
- a procedure P whose body is the ENCLOSED-clause of the access-clause and which accepts the defined-indicators and the postlude procedure Q as parameters.

For access-clauses with more than one module-call, all the postludes must be called before the ENCLOSED-clause returns.

AB43.4.1 MABEL: A Beginner's Programming Language.

P.R. King, G. Cormack, G. Dueck, R. Jung, G. Kusner, J. Melnyk

(Department of Computer Science, University of Manitoba,
Winnipeg, Manitoba R3T 2N2, Canada)

ABSTRACT

This paper presents a preliminary version of an introductory programming language. The design of MABEL is far from frozen, and many of the decisions taken are, at best, tentative. Our hope in presenting the language at this stage is to obtain input from a wider source. Hence we earnestly solicit constructive cricitism, and ask readers to accept the current document in this spirit.

1. INTRODUCTION

MABEL (MAnitoba BEginner's Language) is a programming language for people who have never programmed before. It is a simple, general-purpose language. Hopefully, this does not imply that with MABEL one can only do simple things. Rather, MABEL is intended to provide a simple introduction to the art of programming by assisting the newcomer in the design of sequential algorithms. MABEL is designed to be simple to teach and to use.

The designers received suggestions from a variety of sources, both within the University of Manitoba, from students and instructors alike, and from a number of high school instructors within the Winnipeg School System who were asked to identify areas of difficulty encountered both by themselves and by their students. Each member of the group had a "pet" language (PASCAL, COBOL, ALGOL W, ALGOL 68, PL/1 and SNOBOL), features of which were advanced by its proponent and avidly attacked by others. We also listened (though frequently pretending otherwise) to the comments of colleagues outside the group as features on which we sought opinion were surreptitiously leaked.

Existing beginning languages, such as B_0 and the Toronto SP/k system, were given careful attention.

From this diversity of advice, sometimes helpful but often impossible or derisory, the criteria of §2 were established. This list became our bible, sacrosanct and inviolable, by virtue of which all design decisions were taken and to which all disputes were referred.

The majority of the time spent in actual design was spent in taking three basic decisions, namely the primitive types, the data structuring facilities and the parameter mechanism. These decisions and their rationale will be discussed in §3. Once they had been taken, most of the remainder of the design followed relatively rapidly and easily. There was some hectic infighting over the form of the repetitive construct, but the bloodshed was minimal compared to that occasioned by discussions over the primitive types, for example. The remainder of MABEL, after the three basic decisions, will be discussed in §4.

Personal prejudice began to rear its ugly head when deciding upon the concrete syntax, and the current proposals may suffer in that regard as a result of the occasional compromise decision. Some sample programs appear as an appendix, and a MABEL syntax chart, à la Watt, Sintzoff and Peck, is appended.

2. DESIGN CRITERIA FOR MABEL

The objective of MABEL is to provide as smooth an introduction as possible to the esoteric art of programming. Whether or not the beginner will graduate from his lowly state, and what happens when he does, is not deemed of any great relevance in determining how to effect such an introduction. If MABEL is a good introduction to more complex languages we would regard this as a bonus rather than a result of design.

Although nine criteria are explicitly discussed, several points not given in the list were considered, but ultimately excluded from the design. These included whether MABEL should provide an introduction to machine architecture, whether MABEL should be extensible and whether MABEL should define such (non-elementary but potentially simple) features as program modules and linkage to other languages. Such features are now being considered in the context of a systems implementation language being designed as a MABEL superset.

We first consider five "positive" criteria: those which were of major importance in the design of MABEL.

(i) Simplicity. The beginner must not be confused by a large number of unorthogonal features. There must be no discrepancies between the meaning of constructs when they appear in different situations or in the ways in which they may be used. Thus, the distinction between <statement> and <simple statement> in ALGOL W is not simple; the ALGOL 68 iterative statement is far from simple both because one requires so much information before it can be used even for simple applications, and because two applications, such as

```
      FOR i TO n DO  read(a[i]) OD
```

and

```
      WHILE REAL x;  read(x); x>0 DO SKIP OD
```

have vastly different forms and purposes; the use of pointers and associated dereferencing and aliasing is very far from simple.

(ii) Readability. The reader should find MABEL programs relatively self-documenting and self-verifying. These requirements impinge on design at both the abstract and concrete levels.

MABEL must be surprise free, conform wherever possible to accepted mathematical meaning (5/3 <u>is</u> the same as 5.0/3.0) and adhere to the precepts of structured programming.

(iii) Teachability. No feature was added to MABEL until one had demonstrated a simple means of teaching it to beginners. The language should be teachable in a "continuous" fashion, by incorporating features which can be exemplified and assimilated in small "upwards-compatible" stages, rather than features which require a lot of detailed information before they can be put to simple use.

(iv) Introduction to design of algorithms. The beginning programmer is habitually faced with two problems; firstly, to design a sequential algorithm for the problem at hand, which is rarely in sequential form, and secondly, to cast this algorithm into the form required by the particular programming language being used. MABEL has been designed to assist the user in the first of these: all else is of subsidiary importance, and one will observe that MABEL lacks certain "standard" language features since they do not contribute directly to this end.

(v) Versatility. Many students have a low opinion of their introductory language as a direct result of disappointment in the applicative examples with which the "power" of the language was illustrated. One wonders to what extent such samples are chosen simply because the language in question is just so restricted. MABEL is a general-purpose language and, as the examples in the appendix show, has the power to be used for "real" problems. We hope that MABEL will cater, to some extent at least, to the ex-beginner who nonetheless wishes to continue using MABEL because he likes it.

Two further criteria were deemed of somewhat less importance:

(vi) Small compiler. It is quite probable that a common environment for a language like MABEL will be mini-computers. Thus the MABEL compiler must be of modest size, and this should be reflected in the language design.

(vii) Simple compilation. It is highly desirable that MABEL be 1-pass. Equally, it must be easy to associate clear, meaningful diagnostics with both compile-time and run-time errors. Our experience is that these latter questions are as much matters of language design as of compiler design.

Two final criteria were considered to be of rather minimal importance to the design of MABEL:

(viii) Introduction to programming languages. "*And visit the sins of the fathers upon the children unto the third and fourth generation.*" This theology appears rampant in programming language design (and is, we claim, responsible for a multitude of disastrous design decisions). It is not the philosophy of MABEL: we do not accept that transition from a simple to a more complex language is facilitiated by incorporating bad features from the complex language in the simple one.

(ix) Run time efficiency. Although the run time efficiency of both time and space requirements are of minor importance in the design of a beginner's language, they should not be entirely ignored if the language is to gain any degree of acceptance as a viable product.

3. THREE FUNDAMENTAL DESIGN DECISIONS

i) Simple types in MABEL

MABEL has a single simple type. The programmer may define and manipulate constants and variables of this simple type, and compose structured types from it. In this respect, MABEL resembles SNOBOL 4, and uses the same syntax for literals, representing them as character sequences enclosed within ", " or ',', pairs.* MABEL is not a string processing language. Naturally, the programmer will be aware that certain program variables are restricted to certain subdomains of this one type and that certain operators make sense only in certain subdomains, but MABEL considers such subdomains to be entirely the programmer's responsibility rather than a static, feature. (A clever compiler, however, might handle some of them statically.)

In retrospect one wonders why this decision took us so long to take; it now appears entirely natural and obvious. The reasons for

---

* Currently, the quotes are optional for integer constants. This is under review.

having multiple pre-defined types, in ALGOL for example, appear to be

- increased static security
- increased readability, and self-documentability
- increased run-time efficiency
- ability to use generic operators

The first of these is to a large extent an implementation concern and low on our score-card. The second is highly debatable. One can as easily read and comprehend

```
P + B * Q - 3      or
A AND C OR Q<3
```

without consulting the declarations or knowing the types of A, B, C, P or Q as with; any complete understanding requires detailed dictionary type descriptions in either case. Few beginner programs ever run in production mode; thus the third reason hardly applies. Finally, generic operators are especially confusing to the beginner; why should

```
"PQRS" < "XYZ"      or      3* "XYZ"
```

be meaningful? If one means "comes alphabetically before" or "replicate three times", then one should say so.

Further, multiple types add to the complexity of a language per se, by virtue of the diverse denotations required and, most of all, by virtue of the type conversions, both explicit and implicit. The beginner needs no assistance in accepting that

```
'3' + '4.7'
```

is perfectly sensible and yields '7.7', whereas

```
'3' + 'XYZ'
```

is not sensible and will produce rubbish.

It is to be admitted that typing permits certain errors to be caught earlier than can be done in a typeless language, but it is not clear that the class of such errors is sufficiently broad to justify the complexity of multiple types. On the other hand the adoption of a single type added considerably to the ease of description of transput (c.f. §4 iv) and assisted greatly in defining the data structuring facility of MABEL, the second fundamental design decision.

(ii) Data Structures in MABEL

In order to satisfy the criterion of versatility, MABEL should provide the powers afforded by conventional data structures, including pointers, heap-storage management and flexible arrays. Conventional arrays as in FORTRAN and ALGOL would be quite unorthogonal with the single primitive type of MABEL. Restricting indexation to integers is inappropriate since integer is not a predefined type, and since strings have no inherent order, the concept of an array as an ordered set is equally unsuitable. It was decided to replace the conventional array by a facility such as the table concept of SNOBOL. SNOBOL tables are one-dimensional and each item is selected by a unique key; use of the same key accesses the same element while use of a new key creates a new element.

Next, the possibility of multi-dimensional tables was considered. To examine their usefulness in a beginner's language, illustrative examples currently used for high-school and first-year students were scrutinised. Most examples appeared highly contrived to make use of two dimensional arrays. A typical example is the construction of a table of student numbers and their grades according to course number. There are usually far more courses offered than are taken by a particular student so that the table will typically be sparse; the beginner is then forced to write code (to ignore the empty entries) which is not part of the processing algorithm. What is needed is a table keyed by student-name, with each entry a table of marks keyed by course-name.

From these considerations emerged the MABEL data structure as a table with multiple sub-keys, where a key may be any expression which yields a simple value. A multi-dimensional array would be represented by using the same number of keys at all times. A COBOL or PL/1 structure is achieved by restricting keys to constants. By making use of the full power of an arbitrary number of variable keys, any tree whatsoever may be represented as a MABEL STRUCTURE, without introducing any notion whatsoever of pointers. Some examples illustrating these remarks appear in the appendix; the reader might wish to consult these before continuing.

The following formal rules serve to describe the syntax and semantics of the MABEL STRUCTURE facility:

A A structure may have a simple value or a multiple value. Let S be an arbitrary structure (which might, of course, be a variable or constant or an expression or delivered by a function) and k, k_1, k_2, ... arbitrary keys.

B (i) If S has a multiple value, S may be qualified thus:

S.k

to yield the corresponding (sub-) structure.

(ii) If S has a simple value then S may not be qualified.

C (i) If S has a simple value then S may be explicitly coerced to yield that value thus:

S.

(ii) If S has a multiple value, S may not be so coerced.

Thus, a reference to a sub-structure of S is of one of the forms

S S.k ...... $S.k_1.k_2. \dots .k_n$

while a reference to an element (simple value) in a structure is of one of the forms

S. S.k. ...... $S.k_1.k_2. \dots .k_n.$.

MABEL structures also permit heap-like memory management. Assuming that the MABEL prelude contains a function UNIQUE*, successive calls of which produce distinct, arbitrary simple values, then the following four groups of code contain equivalent phrases:

A. ALGOL 68:

```
     MODE T = STRUCT (REF T link, INT i);
     REF  T   p;
```

PL/1:

```
     DECLARE 1  T BASED,
                2 LINK POINTER,
                2 I FIXED BINARY;
     DECLARE P  POINTER;
```

MABEL:

```
     STRUCTURE   T;
     CONSTANT    LINK:"LINK";           #FIELD OF T. name
     CONSTANT    I:"I";                 #FIELD OF T. name
     VARIABLE    P;                     #NAME WITHIN T
```

B. ALGOL 68:

```
      p : = HEAP T : = (NIL, 17)
```

PL/1:

```
      ALLOCATE  T  SET (P);
      P → T.LINK = NULL;
      P → T.I    = 17;
```

MABEL:

```
      P: = UNIQUE;
   T.P : = (| LINK: NULL, I:17  |)
```

C.

```
ALGOL 68:     link OF p
PL/1    :     P → T.LINK
MABEL   :     T.P.LINK.
```

D.

```
ALGOL 68:  no explicit garbage collection
    PL/1:  DELETE P→T
  MABEL:   T.P.  := UNDEFINED*
```

A typical declaration would be

```
VARIABLE   UNIQUEX:
FUNCTION   UNIQUE  RETURNS VALUE:
    UNIQUEX := UNIQUEX & 'Z';
RETURN UNIQUEX
```

---

* The MABEL prelude contains the declaration of a constant UNDEFINED whose value is "$$UNDEFINED". All MABEL simple variables are initialized to that value (including UNIQUEX used in the preceding footnote).

These examples, together with those in the appendix, illustrate how the MABEL structure facility provides all the power deemed necessary while maintaining its essential simplicity. It will be remarked how central the single simple type is to its formulation. We are grateful to Robert Dewar for pointing out the similarity between the STRUCTURE of MABEL and maps in the language SETL, although the MABEL feature was developed quite independently and with different goals.

(iii) The parameter mechanism in MABEL

It is essential that MABEL have a simple parameter mechanism. Further, the beginner should not be burdened with words like VALUE, RESULT, name, reference and their diverse and confusing effects. MABEL therefore has a single parameter transmission mechanism: all parameters are called by "constant", that is, by value without the "free" local variable. Thus no formal parameter can be assigned to, a natural and readily assimilated rule; to a mathematician, the notion of a function changing one of its arguments is quite foreign.

A mechanism is needed for returning one or several values. In MABEL this is achieved by a RETURN statement.

Notice that structures are passed in the same manner. (The specification of a function includes the specification of each parameter as a structure, function or simple value, the latter being the default, as well as the specification of the value(s) returned). Since copying of structured values is only necessary when the actual parameter is used non-locally in the procedure body and assigned to, and such instances can be easily detected statically, the mechanism is not inefficient. One can optimise further by only copying the entries in the structure which are changed (as is done with multiple values in the ALGOL 68S compiler.

A function or procedure is quite permissible as a parameter; we have endeavoured to make it clear from the syntax that it is the function which is passed and not the value yielded by a call.

4. OTHER MABEL CONSTRUCTS

i) Control structures.

MABEL is range-structured, a new range and scope being defined by either a block or a function (procedure) body.

MABEL has a single conditional construct which, following the philosophy alluded to in §2 (iii) may be introduced incrementally without confusing the beginner. A simple conditional would be

```
IF A
IS B THEN statement
```

which may be supplemented by an else part:

```
IF A
IS B THEN statement 1
ELSE statement 2
```

Both statement 1 and statement 2 may comprise a sequence of statements (in which case, each statement in the sequence will be indented; c.f. §4(v)).

The conditional may be further extended:

```
IF A
IS B THEN statement 1
IS C THEN statement 2
IS D|E THEN statement 3
ISNT F|G|H THEN statement 4
ELSE statement n+1
```

A is compared with B,C,D,E,F, etc. consecutively until a match is encountered; in the case of IS, the corresponding statement is executed, while in the case of ISNT attention is turned to the next comparison if there is one.

There appears to be no problem in teaching this construct. We are encouraged to believe that it is highly readable by virtue of supportive evidence from a series of experiments in which a sequence of examples was presented to a number of non-programmers, none of whom had any difficulty in describing the flow of control.

MABEL has two repetitive constructs. The form of the first is

```
        FOR id INDEXING structure DO
            statement-list
```

where the statement-list will probably involve structure.id. This permits indexing over an entire structure, and is somewhat similar to its counterpart in B_o, although MABEL has no range concept.

This construct is useful but limited. For example, FOR cannot imply an order in which the elements of the structure are accessed. MABEL therefore provides a second, completely general repetition facility, which permits both counting loops and recursive loops. The simplest form is

```
     REPEAT
```

The elaboration consists in replacing REPEAT by a copy of the block in which it occurs. (Notice that REPEAT is not equivalent to a GOTO.) At the head of the block a number of variables may be initialised and they may be updated by REPEAT

```
     BEGIN WITH I; = 1
       .
       .
       .
     IF I ISNT 10 THEN REPEAT WITH I+1
       .
       .
       .
     END
```

Again, this powerful feature is easy to teach. One's first demonstration program is usually

```
    BEGIN
    VARIABLE X,Y,Z;
    GET X,Y;
    Z:= X+Y;
    PUT Z, X, Y;
    END
```

The bright student in the front row usually objects at this point that the program only handles one set of data, and will ask how one may "repeat the process". Upon seeing the program

```
    BEGIN
    VARIABLE X,Y,Z;
    GET X,Y;
    Z:= X+Y;
    PUT Z,X,Y;
    REPEAT
    END
```

the same bright student may press his luck, object that the loop is infinite and wonder how one may "put a limit on the number of times it repeats". (If one does not have a bright student, plant an accomplice.) At this stage, one introduces a simple WITH and conditional. Later on one may examine the effect of instructions between REPEAT and END.

Perhaps one should re-emphasize that MABEL is principally designed to provide an introduction to the formulation of algorithms, a wide class of which are recursive. It thus seems entirely appropriate to include a recursive control structure.

These are currently the only loops in MABEL. A possible drawback is that loops analogous to

```
    FOR i TO UPB a - 1  DO ... OD
```

must be written using REPEAT. From a number of examples the designers feel that this is not a serious drawback; we would not be averse to including a further iterative construct but a satisfactory one has still to be found.

(ii) Subroutines, calls and formulae.

The return statement of a function (the last statement in the function body) may return several values:

```
RETURN  I, J,  (I+J),  ARRAY.I.J.
```

Coupled with this, MABEL permits parallel assignations as in

```
I, J := 3, 4;
A, B := B, A;
X,Y,Z:= A,B MULT C,D;
```

The third of these may not be entirely clear. Many languages distinguish between operators and functions. The language provides both, but the programmer (usually) may only define functions. This implies that formulae involving user-defined operations must be written in Polish notation, which is confusing to both the programmer and reader.

MABEL makes no distinction between an operator and a function. A function in MABEL has an arbitrary number of left and right parameters and returns an arbitrary number of results. A function may have no parameters or zero left parameters, but we feel this latter will occur less frequently than might be presupposed. Not only does this afford a natural way to write functions and calls, but is an excellent aide-memoire. One can more easily remember the specifications of the substring function, for example, if one writes

```
A  SUBSTR  I,J
```

rather than

```
SUBSTR (A,I,J)
```

Thus, MABEL function calls are simply an extension of the familiar notation

```
X + Y
```

The possible syntactic ambiguity in, for example

```
.... := X   PLUS  Y,Z
```

is easily resolved by parenthesising calls in a list, as in

```
A  SUBSTR (I + J), (MAX  K,J)
```

The introduction of user-defined infix operators raises the question of how operator priorities shall be handled. The possibilities appear to be

- no priorities, which would require that formulae would have to be completely parenthesized
- integer priorities, as in ALGOL 68
- a small number (say 4) of priority levels, the priority of a new operator being defined by something like

  ```
  PRIORITY   ADD   LIKE +
  ```
- left-to-right (or similar) evaluation, optionally combined with any of the first three.

Of these possibilities, the second appears the least satisfactory; programmers in general and beginners especially remember relative rather than absolute priorities. The third has attractions, but requires a new construct, requires that a user assign a priority even if he does not wish to for a particular operation, and implies the introduction of somewhat arbitrary decisions; one could argue for days about the relative priorities of things like SUBSTR and & (concatenate), for example. To avoid such arbitrariness MABEL currently uses the first alternative and ignores the fourth, but this is somewhat tentative.

As remarked previously, a function (or procedure) specification involves specification of the parameters and values returned; this is naturally true for function parameters, which are specified using a "model" as in

```
FUNCTION    FUNCTION    F  SIMPSON   A,B  RETURNS VALUE:
   MODEL    F X   RETURNS VALUE;
   VARIABLE S, H, N;
            .
            .
            .
   RETURN (H * ((( F A ) + (F B) ) +S))  /3
```

If F were to have a function or procedure parameter, it too would have a model.

(iii) Declarations and constants

All variables must be declared. All declarations must appear at the head of a block on a function (procedure) body. There is no initialisation of variables within declarations.

These restrictions, if indeed one considers them restrictions, are made for pedagogical reasons, although they also assist in one-pass compilation. Consider the examples:

```
BEGIN                                   BEGIN
VARIABLE C;                             VARIABLE C;
C:=  7 ;                                C: = 8;
   .                                       .
   .                                       .
   .                                       .
    BEGIN                                   BEGIN
    VARIABLE D:=C;                          C: = 7;
         .                                       .
         .                                       .
         .                                       .
    VARIABLE C:= 5;                         VARIABLE C;
         .                                       .
         .                                       .
         .                                       .
```

These are both grossly unreadable and will cause intolerable surprises to the newcomer (if not the expert too!).

Compile time constants are permitted as in

CONSTANT PI:'3·14159', PRIMES: (| '1':2, '2': 3, '3': 5, '4': 7|)

but the following is not permitted

```
     CONSTANT NEWPI:  '4'*(ARCTAN 1);
```

It might be hard to explain to a beginner why NEWPI should be a "constant" and would be hard to prohibit examples like

```
     CONSTANT  A:B,   B:A;
```

in a consistent manner.

iv) Transput

MABEL provides two sets of transput primitives. The first is intended for use by rank beginners, and is an extremely simple stream transput facility. The beginner will, at a very early stage, appreciate the meaning of

```
          X, Y, Z  : = '1', '2', '3';
```

and shortly thereafter will learn that

```
          GET  X, Y, Z
```

where the data contains the list of literals

```
          '1', '2', '3'     (or '1'  '2'  '3')
```

means precisely the same thing. The corresponding output construct is typified by

```
          PUT  (X + '1'),(Y + '1'),(Z + '1');
```

which produces

```
          '2'  '3'  '4'
```

on the printed page. The remaining primitive he may use is

```
               NEWLINE
```

This elementary format-free transput is easy to learn but is insufficient for all but the most basic purposes. MABEL also provides two simple record transput primitives:

```
     READ   var,  var,  var, ..., var;

     WRITE  exprn,  exprn, ..., exprn;
```

which may optionally specify a file-name:

```
     READ  A,B,C  FROM STANDIN;

     WRITE (A+B), (C+D) TO STANDBACK;
```

Each variable is read from and expression written to a new record in the appropriate file, which is STANDIN or STANDOUT (which are also accessed by GET and PUT) if no file name is specified. MABEL takes the view that the beginning programmer should be made aware that transput operations are essentially string transfers; hence, it is the programmer's responsibility to manipulate the corresponding strings as he wishes (although MABEL will provide various functions to assist him).

It will be observed that all these transput operations involve constructs rather than function calls. We consider the additional seven reserved words introduced (for a total of 32) far preferable to introducing "pseudo" functions with a variable number of parameters, as is the case in ALGOL W.

(v) Operations and other oddments.

The MABEL "system" comprises three components: the kernel, the prelude and libraries.

The kernel incorporates all the MABEL constructs, including a set of "primitives" which will rarely be used by programmers but which are complete in that all operations may be defined in terms of them as described in the next paragraph. The primitives currently used are

```
SPLIT char FROM string
APPEND string TO string
```

The kernel includes some global constraints, such as file names but does not include any function or procedure definitions.

In the prelude are defined a host of MABEL functions. These include arithmetic operations such as

+, -, X, /, **, <, > etc., DIV, MOD, FLOOR etc.

string operations such as

& { concatenate}, SUBSTR { "ABCD" SUBSTR 0,2 yields "AB"},

CB, CA { comes alphabetically before and after}, REPLACE,

CONTAINS, REVERSE

and the (non-McCarthy) logical operations

AND, OR, NOT, XOR

The prelude may be written entirely in terms of the kernel, and this will be incorporated in the definition of MABEL. Hopefully this definition of the prelude will be correct and an aid to portability; it should be directly usable by an implementer with possible loss of efficiency being the only penalty.

A number of standard libraries will be included in the MABEL definition. Others may be added at installations.

There are two ways to include comments in MABEL programs.

. All text from # to the end of the current input record is treated as comment.

. Comments may be included within the brackets (*, *).

In the latter case the brackets may be nested and will be matched by the compiler, thus permitting sections of program including comments to be "commented out" for testing purposes.

MABEL currently uses a 56 character set consisting of

. letters A-Z

. digits 0 - 9

. operators + - * / ¬ = < >

. punctuation ( ) ' " ; : . , | # space

ASCII has currently been adopted as the collating sequence. Identifiers are (arbitrarily long) sequences of letters and digits starting with a letter, while function symbols are identifiers or sequences of operators. (Special symbols are never sequences of operators.) We emphasize that generic operators are not permitted; operator (function) identification follows precisely the same rules as for identifiers.

MABEL encourages good program layout. When a group of statements is to form a single compound statement and there are no explicit delimiters, these statements must be indented,at the same level. This applies in three situations: following THEN, following ELSE and following DO. (All other compound statements have delimiters such as BEGIN...END, PROCEDURE....FINISH and FUNCTION....RETURN.) We feel that good program layout should be mandatory rather than optional; indentation is a powerful, all too frequently ignored control structure.

5. MABEL Implementations

A compiler for the current version of MABEL has been written at the University of Manitoba. It could be made available to anybody willing to experiment with the language. Its brief specifications are as follows:

Computer:	IBM 370 under OS or VS
Source Language:	PL/1
Compiler Size:	200 K
Space Requirements:	Compiler: 256 K Run time: 4K + Object Code + Memory area (run-time parameter)
Parser:	LR(1) with local error correction
Object Code:	370 Object Code Object code is combined with 4K of run-time routines. Run time includes a garbage collector and error traceback.

ACKNOWLEDGEMENT

The first-named author wishes to thank his grade 2 daughter for pointing out that 10-3+2 is equal to 5.

APPENDIX: ILLUSTRATIVE EXAMPLES

Three examples are given, all of which have run successfully under the current MABEL compiler. The first is a simple prime seive program; the second evaluates simple arithmetic expressions while the third, a family tree program, is intended to illustrate the power and potential of the MABEL STRUCTURE facility. Two sets of output appear for the third program; the second set illustrates the run time dump produced in the event of a run time error (here activated by execution of the statement STOP).

```
        MABEL COMPILER   VERSION 2   RELEASE 1   (JAN 1977)                          PAGE    1

*OPTIONS SPECIFIED* LMARGIN=0,,RMARGIN=8,PAGES=5,CODEGEN=0,LINES=60

*OPTIONS IN EFFECT* LINES=60,PAGES=005,CODEGEN=0,LMARGIN=00,RMARGIN=08,OBJECT=OBJECT

    003A     1 (* THIS PROGRAM USES THE SEIVE METHOD TO CALCULATE THE PRIME
             2    NUMBERS BETWEEN 1 AND MAX     *)
             3
             4   BEGIN
             5      CONSTANT FALSE : 'FALSE', TRUE : 'TRUE';
    0088     6      CONSTANT ZERO : 0, ONE : 1;
    00B0     7
             8      STRUCTURE SEIVE;
    00B6     9      VARIABLE UNDEFINED;
    00BA    10      CONSTANT MAX : 100;
    00D0    11
            12      BEGIN WITH J := 2;
    0106    13         IF SEIVE.J. IS UNDEFINED THEN
    0120    14                        WRITE J;
    0126    15                        BEGIN WITH K := J + J;
    016E    16                           IF MAX >= K IS TRUE THEN
    01A2    17                                          SEIVE.K. := FALSE;
    01AC    18                                          REPEAT WITH K + J;
    01DA    19                        END
    020E    20         IF MAX > J IS TRUE THEN REPEAT WITH J + ONE;
    027C    21      END
    02B0    22   END

  NO SYNTAX ERRORS
  NO SEMANTIC ERRORS

2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

```
      MABEL COMPILER    VERSION 2   RELEASE 1    (JAN 1977)                              PAGE  1

*OPTIONS SPECIFIED* LMARGIN=0,,RMARGIN=8,PAGES=5,CODEGEN=0,LINES=60

*OPTIONS IN EFFECT* LINES=60,PAGES=005,CODEGEN=0,LMARGIN=00,RMARGIN=08,OBJECT=OBJECT

   003A        (*
                    THIS PROGRAM READS EXPRESSION STRINGS AND EVALUATES THEM.
                    STRINGS ARE OF THE FORM (3+(4*5)). THE OPERATORS ALLOWED
                    ARE +,-,*,/. BLANKS ARE IGNORED.
                    EXPRESSIONS MUST BE FULLY PARENTHESIZED.
                                                                                 *)
               BEGIN
                  CONSTANT NULL : '';
   006C           CONSTANT BLANK : ' ';
   0080           FUNCTION A DIV B RETURNS VALUE:
   00AA              VARIABLE Q,R;
   00B2              Q,R := A DIVREM B;
   00E4              RETURN Q;
   0118           FUNCTION +-*/ L,OP,R RETURNS VALUE:
   014C              VARIABLE RESULT;
   0150              IF OP
                     IS '+' THEN
   017C                 RESULT := L + R;
   019E              IS '-' THEN
   01D6                 RESULT := L - R;
   01F8              IS '*' THEN
   0230                 RESULT := L * R;
   0252              IS '/' THEN
   028A                 RESULT := L DIV R;
   02AC              IS NULL THEN
   02D0                 RESULT := L;
   02D6              RETURN RESULT;
   0304           FUNCTION EVAL STRING RETURNS VALUE:
   0324              VARIABLE L,OP,R,VAL,CH;
   0338              L,OP,R,VAL := NULL,NULL,NULL,STRING;
   0350              BEGIN
                        SPLIT CH FROM VAL;
   0374                 IF CH
                        IS '*'|'-'|'/'|'+' THEN
   0412                    OP := CH;
   0418                    REPEAT
                        IS '(' THEN
   0450                    VAL := EVAL VAL;
   046C                    REPEAT
                        IS ')' THEN
   04AA                    VAL := (+-*/ L,OP,R) & VAL;
   04EA                 IS NULL THEN
   050E                    VAL := ')';
   0522                    REPEAT
                        IS BLANK THEN
   054C                    REPEAT
                        ELSE
   0558                    IF OP IS NULL THEN
   0570                             L := L & CH;
   0592                    ELSE
   059E                       R := R & CH;
```

AB 43p.77

```
    MABEL COMPILER   VERSION 2   RELEASE 1   (JAN 1977)                     PAGE   2
   05C0                     REPEAT
   05C6                  END
   05EE               RETURN VAL;
   0622

                   BEGIN                      # MAINLINE
                      VARIABLE INPUT;
   0642               READ INPUT;
   0648               IF INPUT ISNT 'END' THEN
   0670                          WRITE INPUT & ('=' & (EVAL INPUT));
   06CA                          REPEAT
                      END
   070A            END

 NO SYNTAX ERRORS
 NO SEMANTIC ERRORS


1+2=3
1+(2*3)=7
((1 + 2)*(4 + 7))/7=4
    0.04 SECONDS EXECUTION TIME
       0 STORAGE REGENERATIONS
```

```
MABEL COMPILER   VERSION 2   RELEASE 1   (JAN 1977)                              PAGE   1

*OPTIONS SPECIFIED* LMARGIN=0,,RMARGIN=8,PAGES=5,CODEGEN=0,LINES=60

*OPTIONS IN EFFECT* LINES=60,PAGES=005,CODEGEN=0,LMARGIN=00,RMARGIN=08,OBJECT=OBJECT

003A      (***************************************************************

             FAMILY TREE PROGRAM

           ****************************************************************)

          BEGIN
          VARIABLE UNDEFINED;
005A      CONSTANT NULL: '';
0070      STRUCTURE FAMILYTREE;
0076         CONSTANT MOTHER: "MOTHER";
008E         CONSTANT FATHER: "FATHER";
00A6         CONSTANT CHILD:  "CHILD";
00BE         CONSTANT SIBLING: "SIB";
00D4            # PARENTS NAME FURTHER QUALIFIES SIBLING;
          PROCEDURE BIRTH  NAME,NEWFATHER,NEWMOTHER:
0108         VARIABLE MIDDLENAME;
010C         IF FAMILYTREE.NAME.FATHER.
             IS UNDEFINED THEN
0128            FAMILYTREE.NAME :=
                   (| MOTHER:  NEWMOTHER,
0140                  FATHER:  NEWFATHER,
014A                  SIBLING: (| NEWMOTHER: FAMILYTREE.NEWMOTHER.CHILD.,
0166                              NEWFATHER: FAMILYTREE.NEWFATHER.CHILD.|),
017E                  CHILD: NULL |);
0192            FAMILYTREE.NEWFATHER.CHILD. := NAME;
019E            FAMILYTREE.NEWMOTHER.CHILD. := NAME;
01AA         ELSE
01BC            WRITE "NAME " & (NAME & "IS NOT UNIQUE; SPECIFY A MIDDLE NAME");
0234            GET MIDDLENAME;
023A            PUT MIDDLENAME; NEWLINE;
0244            CALL BIRTH (NAME & MIDDLENAME),NEWFATHER,NEWMOTHER;
0274         FINISH;
02A2      PROCEDURE ANCESTORS  NAME,PRTLINE:
02CC        WRITE PRTLINE & NAME;
02EE        IF FAMILYTREE.NAME.FATHER.
            IS NULL THEN
030A           WRITE (PRTLINE & "     ") & "FATHER UNKNOWN";
036C        ELSE
037E           CALL ANCESTORS FAMILYTREE.NAME.FATHER.,(PRTLINE & "     ");
03C0        IF FAMILYTREE.NAME.MOTHER.
03C6        IS NULL THEN
03E2           WRITE (PRTLINE & "     ") & "MOTHER UNKNOWN";
0444        ELSE
0456           CALL ANCESTORS FAMILYTREE.NAME.MOTHER.,(PRTLINE & "     ");
0498        FINISH;
04BA      PROCEDURE OFFSPRING  NAME,PRTLINE:
04E4         WRITE PRTLINE & NAME;
0506         BEGIN WITH NAME,PARENT,PRTLINE :=
                             FAMILYTREE.NAME.CHILD., NAME, (PRTLINE & "     ");
057E            IF NAME
```

AB 43p.79

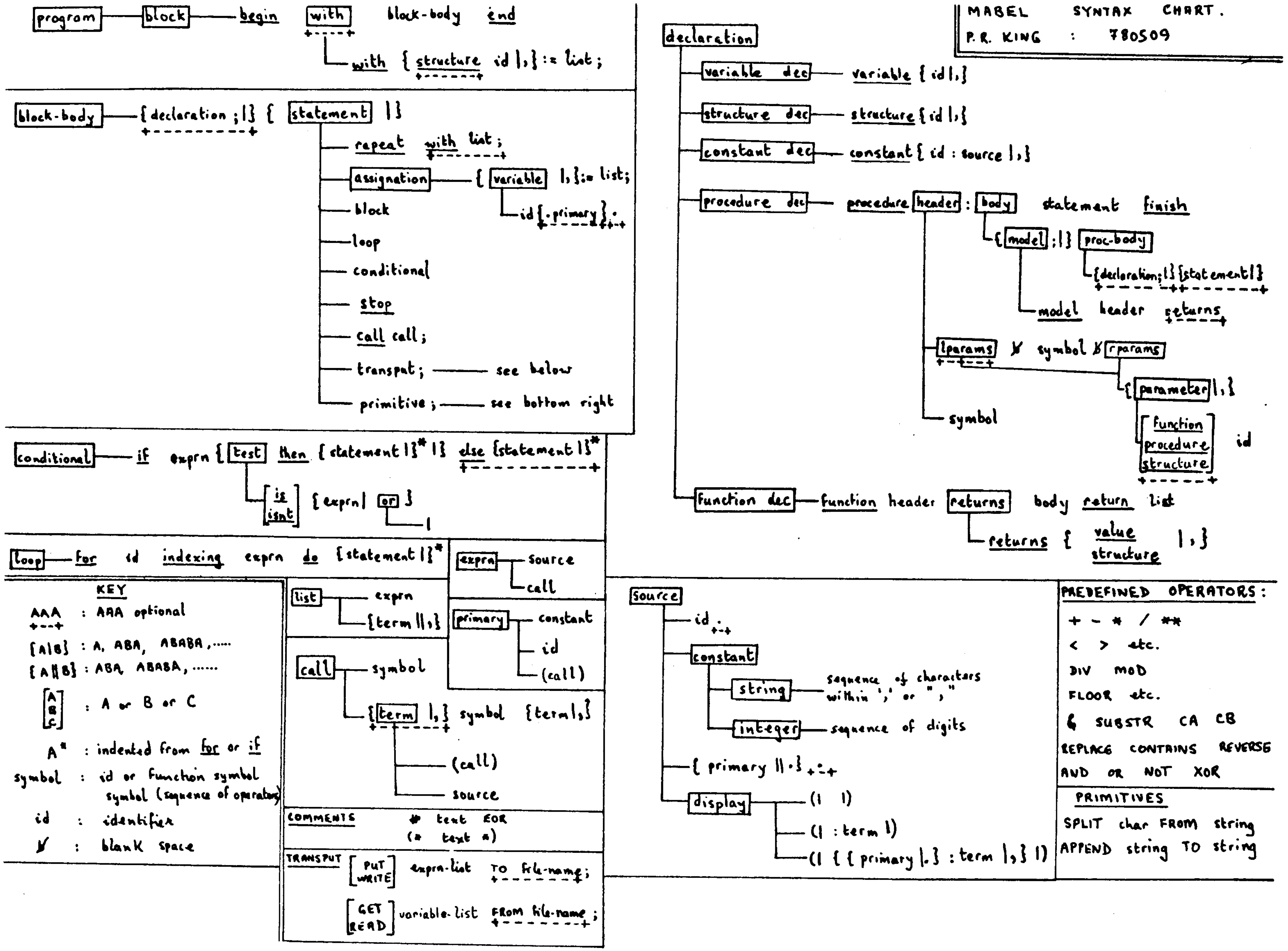
```
     MABEL COMPILER   VERSION 2   RELEASE 1   (JAN 1977)                    PAGE   2
          52         ISNT NULL THEN
  0590    53            CALL OFFSPRING NAME,PRTLINE:
  05A2    54            REPEAT WITH FAMILYTREE.NAME.SIBLING.PARENT., PARENT, PRTLINE;
  05C2    55         END
  05FC    56      FINISH;
  0618    57 PROCEDURE SIBLINGS NAME:
  0638    58      FOR PARENT INDEXING FAMILYTREE.NAME.SIBLING DO
  065E    59         WRITE "SIBLINGS WITH PARENT: " & PARENT;
  06A0    60         BEGIN WITH NAME,PARENT := FAMILYTREE.PARENT.CHILD., PARENT:
  06E4    61            IF NAME
          62            ISNT NULL THEN
  06F6    63               WRITE NAME;
  06FC    64               REPEAT WITH FAMILYTREE.NAME.SIBLING.PARENT., PARENT;
  0716    65            END
  074A    66      FINISH;
  076C    67 VARIABLE COMMAND,P1,P2,P3,P4,P5;
  0784    68 FAMILYTREE := (| NULL: (|MOTHER:NULL,FATHER:NULL,CHILD:NULL|) |);
  07D0    69 BEGIN
          70      GET COMMAND;
  07F2    71      PUT 'COMMAND:',COMMAND;
  0812    72      IF COMMAND
          73      IS "END" | UNDEFINED THEN PUT "GOOD BYE......";
  0872    74      IS "BIRTH" THEN
  08AE    75         GET P1,P2,P3;
  08C0    76         PUT P1,P2,P3; NEWLINE;
  08D6    77         CALL BIRTH P1,P2,P3;
  08EE    78         REPEAT
          79      IS "ANCESTORS" THEN
  092E    80         GET P1;
  0934    81         PUT P1; NEWLINE;
  093E    82         CALL ANCESTORS P1,NULL;
  0950    83         REPEAT
          84      IS "OFFSPRING" THEN
  0990    85         GET P1;
  0996    86         PUT P1; NEWLINE;
  09A0    87         CALL OFFSPRING P1,NULL;
  09B2    88         REPEAT
          89      IS "BROTHERS" | "SISTERS" THEN
  0A1C    90         GET P1;
  0A22    91         PUT P1; NEWLINE;
  0A2C    92         CALL SIBLINGS P1;
  0A38    93         REPEAT
          94      IS "STOP" THEN STOP
  0A78    95      ELSE
  0A7E    96         PUT "INVALID COMMAND, RE ENTER:";
  0AAA    97         REPEAT
          98      END
  0ADE    99 END

NO SYNTAX ERRORS
NO SEMANTIC ERRORS
```

```
COMMAND:  BIRTH      GOD
COMMAND:  BIRTH      ABEL       GOD        GOD
COMMAND:  BIRTH      CAIN       GOD        GOD
COMMAND:  HOWDY      INVALID COMMAND, RE ENTER:    COMMAND:  BIRTH      ADAM       ABEL       CAIN
COMMAND:  BIRTH      EVE        ADAM       CAIN
COMMAND:  ANCESTORS EVE
EVE
     ADAM
          ABEL
               GOD
                    FATHER UNKNOWN
                    MOTHER UNKNOWN
               GOD
                    FATHER UNKNOWN
                    MOTHER UNKNOWN
          CAIN
               GOD
                    FATHER UNKNOWN
                   'MOTHER UNKNOWN
               GOD
                    FATHER UNKNOWN
                    MOTHER UNKNOWN
     CAIN
          GOD
               FATHER UNKNOWN
               MOTHER UNKNOWN
          GOD
               FATHER UNKNOWN
               MOTHER UNKNOWN
COMMAND:  OFFSPRING GOD
GOD
     CAIN
          EVE
          ADAM
               EVE
     ABEL
          ADAM
               EVE
COMMAND:  BROTHERS  EVE
SIBLINGS WITH PARENT: CAIN
EVE
ADAM
SIBLINGS WITH PARENT: ADAM
EVE
COMMAND:  END        GOOD BYE......
     0.06 SECONDS EXECUTION TIME
        0 STORAGE REGENERATIONS
```

```
COMMAND:  BIRTH     GOD
COMMAND:  BIRTH     ABEL      GOD       GOD
COMMAND:  BIRTH     CAIN      GOD       GOD
COMMAND:  HOWDY     INVALID COMMAND, RE ENTER:    COMMAND:  BIRTH     ADAM      ABEL      CAIN
COMMAND:  BIRTH     EVE       ADAM      CAIN
COMMAND:  STOP
*** ERROR *** USER REQUESTED STOP
    PROGRAM WAS EXECUTING AT OFFSET 0A74 IN PROCEDURE MABEL
*** VARIABLES ACTIVE AT TERMINATION ***
VARIABLE UNDEFINED : "$$UNDEFINED"
CONSTANT VALUE NULL : ""
STRUCTURE FAMILYTREE : (|"":(|"MOTHER":"","FATHER":"","CHILD":"GOD"|),
                        "GOD":(|"MOTHER":"","FATHER":"",
                               "SIB":(|"":""|),"CHILD":"CAIN"|),
                        "ABEL":(|"MOTHER":"GOD","FATHER":"GOD",
                                "SIB":(|"GOD":""|),"CHILD":"ADAM"|),
                        "CAIN":(|"MOTHER":"GOD","FATHER":"GOD",
                                "SIB":(|"GOD":"ABEL"|),"CHILD":"EVE"|),
                        "ADAM":(|"MOTHER":"CAIN","FATHER":"ABEL",
                                "SIB":(|"CAIN":"","ABEL":""|),"CHILD":"EVE"|),
                        "EVE":(|"MOTHER":"CAIN","FATHER":"ADAM",
                               "SIB":(|"CAIN":"ADAM","ADAM":""|),"CHILD":""|)|)
CONSTANT VALUE MOTHER : "MOTHER"
CONSTANT VALUE FATHER : "FATHER"
CONSTANT VALUE CHILD : "CHILD"
CONSTANT VALUE SIBLING : "SIB"
VARIABLE COMMAND : "STOP"
VARIABLE P1 : "EVE"
VARIABLE P2 : "ADAM"
VARIABLE P3 : "CAIN"
VARIABLE P4 : "$$UNDEFINED"
VARIABLE P5 : "$$UNDEFINED"
    0.10 SECONDS EXECUTION TIME
       0 STORAGE REGENERATIONS
```

MABEL SYNTAX CHART.
P.R. KING : 780509



program — block — begin with block-body end
with { structure id |,} := list ;
block-body — {declaration ;|} { statement ||
repeat with list ;
assignation — { variable |,}:= list;
id {.primary}.
block
loop
conditional
stop
call call;
transput; — see below
primitive; — see bottom right
conditional — if exprn { test then { statement |}* |} else {statement |}*
is
isnt
{ exprn | or }
|
loop — for id indexing exprn do {statement |}*
exprn — source
call
list — exprn
{term ||,}
primary — constant
id
(call)
call — symbol
{ term |,} symbol {term|,}
(call)
source
declaration
variable dec — variable { id |,}
structure dec — structure { id |,}
constant dec — constant { id : source |,}
procedure dec — procedure header : body statement finish
{ model ;|} proc-body
{declaration;|} {statement|}
model header returns
lparams ✗ symbol ✗ rparams
{ parameter |,}
function
procedure
structure
id
symbol
function dec — function header returns body return list
returns { value structure |,}
source — id
constant
string — sequence of characters within ',' or ","
integer — sequence of digits
{ primary ||.}
display — (| |)
(| : term |)
(| { { primary |.} : term |,} |)

KEY

AAA (optional, marked +--+)	: AAA optional
{A|B}	: A, ABA, ABABA, ...
{A||B}	: ABA, ABABA, ...
[A B C]	: A or B or C
A*	: indented from for or if
symbol	: id or function symbol symbol (sequence of operators)
id	: identifier
✗	: blank space

COMMENTS # text EOR
(* text *)

TRANSPUT
[PUT WRITE] exprn-list TO file-name;
[GET READ] variable-list FROM file-name;

PREDEFINED OPERATORS:
+ − * / **
< > etc.
DIV MOD
FLOOR etc.
& SUBSTR CA CB
REPLACE CONTAINS REVERSE
AND OR NOT XOR

PRIMITIVES
SPLIT char FROM string
APPEND string TO string

AB43.4.2

PROPERTIES, FEASIBILITY AND USEFULNESS OF A LANGUAGE FOR PROGRAMMING PROGRAMS

M. Sintzoff
MBLE Research Lab., Brussels

Manuscript prepared for IFIP WG 2.1, August 1978.

Foreword

At the last meeting, in December, it was suggested "That WG 2.1 investigates the properties, feasibility and usefulness of a language helping the specification and the construction of good algorithms". The aim of this highly informal note is to present half-baked ideas on that subject, as requested by our chairman : remember I am not working now on it, but rather on quite limited and technical problems underlying it; this is safer and wiser for me.

Properties : the blue sky

One should be able to write programs which formalize high-level, well-structured and successful strategies for the development and the (trans)formation of programs, on the basis of given specifications or preliminary programs. The primary design goal must be to enhance and to communicate programming techniques.

Such a language must be nicely organized, consistent, simple. The target sublanguage into which the programmed programs are to be expressed has to be extremely clean and systematic; otherwise, there is no hope of a disciplined working-out of programs. To quote P. Naur, from his history of Algol 60 : we need a "scientific and philosophical orientation towards the definition of a useful, elegant and artistic object." The whole language should be a vehicle for teaching, communicating and formulating methods and techniques of programming. And each programming program must be executable by computer.

If any, except this one, of the required properties cannot be ensured, then the design must be dropped.

Feasibility : the doubt

Sure enough, we are not facing the problems of long-range artificial intelligence : the goal is to express the existing knowledge of programming experts, and not to robotize them. Nevertheless, it could well be dangerous and premature to integrate programming within programs : this could be too much, too soon. Do we know enough in problem description, design methods and programming techniques ?

There is another difficulty : how to take into account diverse methods and techniques, various application-areas and the expected hardware. Again, a sensible way out is to unearth clean, logical, mathematical architectures : we are back to the requirement of extreme simplicity, clarity and economy, i.e. ascetism.

Any elaboration on any available language, appears to be inadequate because none has been initially and mainly designed for expressing techniques of program development. The not-invented-here complex should of course be avoided; but we should also remember that twenty years have been necessary for getting if-then-else in fortran.

The feasibility problem is thus a creation problem. Illusions are probable.

Usefulness : the wish

In his pamphlet "Why programming languages are obsolete", T. Winograd writes : "Higher-level programming systems can provide the means to understand and manipulate complex systems and components". In fact, the main axis of useful work, during the last ten years, in the software area, has been the discovery and dissemination of methods and techniques of programming. However, problems do remain : supposedly constructive techniques still require much ingenuity, intuition, intelligence or invention, behind the scenes; noncommunicating subcultures coexist without mutual benefit; requirements and specifications are not easily formulated; adequate methods are not used as well as they should. Hence, to express design techniques in a high-level, formal, constructive language would not be harmful : we would then have a support for thinking, a medium of communication, and a guarantee of realizability. There have been successful designs of good software systems, but most often these designs remained unformulated, inaccessible and isolated, in a sort of unconscious realm. The design process as such is at least as important as the final, finished product.

This approach would also have a beneficial side-effect on the structure of the target programs to be produced : in the search of basic constructs, the constraints of program design provide a better guidance than formal semantics, implementation methods, verification techniques, or intuition.

Here are now a few case studies of some aspects which deserve attention. They have been selected intuitively and too rapidly.

Functions to be programmed

Here is a rather haphazard collection :

- EXPLICIT implicit specs USING method 3
- REPRESENT set BY balanced trees
- APPROXIMATE TEST OF bounds BY linear relations IN proc 2
- SUBSTITUTE def 2 INTO spec 3
- DEFINE info 3 AS info 1 WITH(OUT) info 2
- ENSURE INVARIANCE OF requirement 4 IN module 3
- APPLY reductions 2 TO def 1
- ENSURE TERMINATION OF proc 2 BY variant fn 2
- CREATE prog 3 FROM prog 1 USING arrays FOR lists
- DISTRIBUTE TERMINATION IN communicating syst 2 USING tree order 3

How can we organize the composition or decomposition of specifications, theories, strategies and tactics to be used ? How do we program these functions ? What more primitive constructs are adequate for expressing them ?

Underlying structures :

- Pure expressions, as in combinatory logic without variables.
- Elementary set theory.
- Recursive equations, plus cartesian product, plus pattern matching.
- Production systems.
- Predicate, or other, logic.
- Synchronized communication.
- Horn clauses.

Should a unique underlying structure be used or not ? What application areas must be considered ? How are they covered ?

Unification by systems of rules

Systems of rules with the structure "IF condition THEN action or transformation" adequately model (i) expert knowledge in specific application fields, (ii) concurrent programs, (iii) guarded commands (iv) program transformation systems. Is the integration feasible ?

Unification by input-output

	IN	OUT
logic	precondition	postcondition
accesses	imports	exports
data	parameters	results
communication	receive	send
synchronization	wait	signal
syntax	begin	end

Logic-flow programming is thus an attractive avenue too.

Unification by approximations

Approximation is the foundation of techniques such as weak or abstract interpretation, or as partial evaluation.

Types	Ideal	Approximation
Viewpoints	Strongest, Unfeasible	Feasible, Weaker than ideal, stronger than nothing
assertions	automatic proving	modes
requirements on bounds, scopes, accesses	full check	sufficient static tests
precomputation	perfect optimization	partial elaboration
pattern matching	matching by oracle	pattern filters

The use of this line of thought for the design of programs is still a research problem.

<u>Unification of definition methods</u>

F1 (progr, initial env) = final env

F2 (progr, precondition) = postcondition

F3 (progr, data) = results

F4 (progr, data) = F42 (F41 (progr), data)

F1 : mathematical semantics.

F2 : predicate transformer; abstraction of F1.

F3 : interpreter; constructive model of F1.

F41 : compiler; static approximation (?) of F3.

However, even unified, these definitions remain inadequate. One should replace the excessively detailed definitions of mathematical functions F1 by the corresponding abstract specifications. The verification rules F2 should be replaced by construction functions G2 (precondition, postcondition) = progr. Finally, the definition which is closest to programming methods must be the main one, the others being derived and auxiliary. But, what is the semantics of a programming method ?

Why should we, and how could we, master the complexity of designing a useful language for expressing the very design of useful programs ?

Formation, Transformation, Eternal Mind's
eternal recreation.

Goethe, Faust, Part 2

Postscriptum, after the Jablonna meeting

During the welcoming party, I asked Dorota "How do you program?". She answered "It is simple : I am given a problem, and I solve it". This clearly explains why many feel the striving for a formal expression of program construction could be ill directed and fruitless : there is no theory of problem solving, no discipline formalizing problem solving methods. Of course, one may then limit oneself to a tractable domain. But we know how easily our attention can be restricted to toy properties of toy examples in toy systems. Aren't we too incompetent?

In his Turing award lecture, J. Backus wishes that "Algebraic transformations and proofs use the language of the programs themselves, rather than the language of logic, which talks about programs". Yet, who knows of a good program calculus including, a.o., a workable algebra of concurrency? On the other hand, the use of natural language for constructing programs does not seem quite adequate : every effective science has created its own language in order to help thinking, communication and technical work. Natural language has been abandoned as a means for expressing solutions to numerical problems. Shouldn't we do the same in the case of program design itself?

Thanks to, and in spite of, various reactions, I still feel the following question should be investigated : how to formulate the design structure of each program, starting from the requirements and the specifications, in the form of a well-designed, well-composed, correct and executable program. A good algorithmic notation should be a vehicle for a good program calculus. We should work along the vertical dimension as well as the horizontal one :

```
    O
    T
    C
    U
ABSTRACTO
    T
    S
    N
    O
    C
```

AB43.4.3 ABSTRACTOd Thoughts.

Hendrik Boom, Mathematisch Centrum, 2e Boerhaavestraat, Amsterdam

Abstract

Some design aspects of a language intended for verified programs are discussed in relation to transformational programming. The data type system of such a language can be intimately related to intuitionist logic.

Computing Reviews categories: 4.20, 4.29, 5.24.
AMS-MOS classification: 68A-30, 68A-40, 02C15.

Key words and phrases: Abstracto, transformational programming, verification, data types, intuitionism, propositional calculus.

1. FOREWORD

In the last few years, Working Group 2.1 has started investigating the transformational approach to programming, in which the programming process is seen as starting with some correct program which expresses some computation in a clear language. By applying semantics-preserving transformations, this program is then converted to a form which can be efficiently implemented on a conventional machine. This is not the same as the more traditional stepwise refinement process, which operates by continually implementing previously unimplemented features. The transformation approach can better be viewed as analysis, translation, and optimization than as refinement. Throughout the transformation process, the program is expressed in an at present undefined language called "Abstracto". Programs in Abstracto need not be directly executable; it is envisaged that Abstracto will contain implementable and unimplementable features, as well as specifications and other useful things. The aim is to support the programming process from first conception of algorithm to final expression as executable code (in "Concreto"). There has also been talk of a language "Transformo" in which to guide the transformation process. The discussions on these points are still extremely vague.

This is a working paper presented at the WG 2.1 meeting at Jabłonna. It is deliberately speculative, and was written to suggest ideas, not to present conclusions of completed research. This may account for a certain half-bakedness in the style of presentation. Minor changes have been made to make it more accessible to the uninitiated reader.

2. OLYMPO

This paper investigates what language can be at the top of the hierarchy. Let us call it Olympo, to make it a highest-level member of the -o family of languages. Olympo should be designed principally to make verification easy, and efficiency be hanged. Perhaps efficiency can be transformed in after, but then we are speaking of Abstracto or even Concreto instead of Olympo.

The current state of the art in formal verification techniques is at present very dependent on automatic theorem proving. The normal approach seems to be to probe a program with assertions much as an acupuncturist inserts strategic pins into a patient, to look at the reaction, and to pass the resulting verification conditions to an automatic verifier. The verification conditions tend to be extremely complicated, probably because attempts are made to let the assertion language mimic faithfully all the convolutions of operational semantics.

An additional source of complexity is that the present-day formal proof and assertions are variations of the n-th order predicate calculus for small n. Predicate calculus was not originally constructed for the practical verification of practical theorems. It was instead viewed as a minimal set of axioms sufficient for the formalization of mathematics, so that the necessary use of logic in mathematics can be precisely studied. Serious attempts to use predicate calculus to formalize mathematics involve massive use of abbreviation. This should be a warning. Present-day assertion languages should be compared to machine languages, and we should seriously attempt to raise their level. A concerted attack on this problem may yield the same kind of progress as we have achieved in high-level languages over the last thirty years.

3. INTEGRATED VERIFICATION

Since the advent of Structured Programming (fanfare on trumpets), there has been a growing suspicion that the correctness proof of a program should be constructed at the same time as its code. Much of the present work on verification ignore this point; in particular, it is ignored in any design for a compiler that accepts a program, generates object code, and also generates verification conditions to be checked (or not) by a separate verifier.

The most successful form of automatically verified assertions to date has been strong data typing. It is extremely rare for a programmer to be tempted to leave out the data types in a strongly-typed language because they involve too much work, or because he can see that the program is correct anyway. The data types are an integral part of the program, and not merely an add-on feature. The temptation to omit assertions is very strong with other mechanical verification schemes. These are clearly add-on techniques. We may conclude:

> The assertion mechanism of Olympo should be so well-integrated into the language that no one would think of leaving out an assertion in the hope of getting a program ready fast.

Each programming language feature should simultaneously generate object code and a proof. It may well be that there may be a variety of features with the same object code but different proof rules. Each would be used in a different situation, and be chosen to make a different application easy to prove. For example, consider the ordinary integral for-loop:

```
for i from 1 to n do S(i) od,
```

where S(i) is a parameterized statement. Various axioms may be convenient to verify programs involving such a loop:

(1) Axiom "for1"

```
For i in 1..n,
   Let V(i) be a set of variables,
   Let S(i) involve only variables in V(i)
   Let P(i) be a predicate.
   Let P(i) involve only variables in V(i)
   Let V(i) and V(j) be disjoint for i ≠ j
   Let {true} S(i) {P(i)}
Then
   {true} for i from 1 to n do S(i) od {FORALL i in 1..n: P(i)}
```

(2) Axiom "for2"

```
for i in 0..n, let P(i) be a set of predicates.
for i in 1..n, let {P(i-1)} S(i) {P(i)}.
Then {P(0)} for i from 1 to n do S(i) od {P(n)}
```

(3) etc.

It is true that axiom (1) can be derived from axiom (2). However, axiom (1) is much easier to use than axiom (2), and is often sufficient. If loops with different axioms were to have different syntax, verification might be simplified for common constructions such as:

```
for i from 1 to n do A[i] := 0.0 od.
```

With the hint that axiom (1) suffices, this loop can be verified (postcondition FORALL i in 1..n A[i]=0.0) without the trouble of inventing an induction hypothesis.

There are probably no more than 10 to 30 commonly used loop structures. It would not be hard to include each of them in a programming language, perhaps as part of a standard prelude. Each of the above for-loop constructions can be considered as a procedure call. The procedure is the for-loop axiom, and its arguments are the pieces of program text "1", "n", "lambda i: S(i)", and the assertions that these pieces of program have specific properties. (Alternatively, these assertions may be considered to be included in the data types. More about this later.)

4. BOUND VARIABLES IN MODES

Consider a procedure which accepts an integer i and yields an array of size i. In Algol 68 we would be able to write its mode as PROC(INT)[]REAL. But why not PROC(INT i)[1:i]REAL? This involves a bound variable in the mode, but provides more information [3].

Bound variables within data types become more useful if one permits the parameters to be modes, and permits their modes to depend on previous parameters. The traditional example is:

```
PROC sort = (MODE M, REF[]M arr, PROC(M,M)BOOL less) VOID:
   C sort the array 'arr' according to the ordering 'less' C.
```

Such formal mode parameters have been traditionally called "modals" in WG 2.1 [2].

5. DATA TYPES AS PROPOSITIONS

It is possible to use the conventional concept of data type (or mode) as extended above to encode the intuitionistic propositional and predicate calculi. Propositions are represented by data types (and not by values of some fixed data type).

A proposition P, when encoded as a data type P, can be considered as the data type of all its proofs. A value of mode P then represents a proof of P. The existence of a value of such a type is equivalent with the provability of the proposition. To make this work, it is of course necessary to abolish facilities like SKIP and NIL, which produce fake values of an arbitrary mode; otherwise everything would become provable. Nonterminating procedures must also be abolished.

The implication P => Q provides a means of converting proofs of P into proofs of Q. We therefore encode P => Q as PROC(P)Q. A proof of P => Q is a procedure which will turn any proof of P into a proof of Q.

We can now construct procedural proofs of tautologies such as P => ((P => Q) => Q). To avoid notational confusion below, we shall use the word PROC as in Algol 68 when we are constructing a mode, but shall write the word LAMB in front of every routine-text. P => ((P => Q) => Q) is represented by the mode PROC(P)PROC(PROC(P)Q)Q. To prove P => ((P => Q) =>

Q), we must construct a procedure of this mode, that is, a routine text starting

```
   LAMB(P a)PROC(PROC(P)Q)Q : ...
```

This accepts "a", a proof of P, and produces a proof of (P => Q) => Q. The "..." must be a routine text of mode PROC(PROC(P)Q)Q, so we may write

```
   LAMB(P a)PROC(PROC(P)Q)Q :
      LAMB(PROC(P)Q ab) Q : ...
```

But now it is easy to obtain an object of mode Q {i.e a proof of Q} by writing ab(a) {combining the proof ab and the proof a}, thus:

```
   LAMB(P a)PROC(PROC(P)Q)Q :
      LAMB(PROC(P)Q ab)Q : ab(a).
```

The call ab(a) corresponds to the use of modus ponens.

It is clear that Algol 68 routine texts are not the best vehicle for expressing proofs. Conventional proofs may even be a better notation for some Algol 68 programs. Some combination of the two may eventually turn out to be appropriate.

The conjunction P AND Q is represented by

```
   MODE P AND Q = STRUCT(P first, Q second).
```

That is, to construct a proof of P AND Q one must combine the separate proofs of P and Q. Furthermore,

```
   MODE P OR Q = UNION(P, Q),
   MODE FALSE = (PROC(MODE M) M),
   MODE NOT P = (P => FALSE).
```

With these definitions, it is possible to find procedures that prove all of Heyting's axioms [1] for the propositional calculus (see the appendix).

Typed universal quantification is straightforward:

```
   MODE FORALL i:T P(i) = PROC(T i) P(i),
```

using the parameterized procedure-yield modes we have already seen. Typed existential quantification can be done with parameterized unions, but we can do it with modals instead:

```
   MODE THEREEXISTS i:T  P(i) =

      PROC(MODE R, FORALL i:T (P(i) => R) ) R.
```

6. IMPLICATIONS FOR PROGRAMMING LANGUAGES

The above results strongly suggest that it is not necessary to separate the assertion language from the data type system, nor to separate the proof language from the procedural language. It may even be undesirable, since the possibilities that may be opened by the direct execution of proofs of theorems are still largely unexplored.

7. FEATURES IN AND OUT OF OLYMPO

Olympo must have facilities for the free construction of programs out of components. The components must be expressible with a high degree of abstraction in order to increase their applicability. Furthermore, there must be no facilities which could lead to lurking side effects, since these would greatly complicate verification.

Therefore:

NO
- side effects
- assignments,
- variables
- GO TOs
- explicit input/output.

These restrictions may well be too severe for anything resembling normal programming; they are introduced so that we can have an easier problem to solve before we start on a hard one. The solution to the easy one may show us how to avoid the hard one entirely.

MAYBE
- heuristic choice and backtracking.

YES
- procedures accepting multiple arguments and yielding multiple results.
- arbitrary typed parameters.
- lots of data types.
- procedures and programs as parameters.
- free algebra with induction law as a primitive data type constructor.
- well-ordering in some form (for termination)
- identity declarations
- promissory notes (see below).

A "promissory note" is a promise that in a later version of a program, some component will be provided that is now missing. The compiler can proceed with syntactic and semantic checks on the rest of the program. Proofs, code, types, and assertions may all be missing in this sense. It is clear that the program cannot be expected to run until essential parts are provided.

It should also be possible to specify "undefined" types and predicates. It may be too much work (or even impossible) to formally write down the complete definition of some predicate or proof. An undefined predicate can then be useful, provided that there is some mechanism for letting the programmer claim that it is satisfied. The compiler can then propagate the assertion throughout the program as part of a data type or otherwise, and can test whether it has been duly claimed whenever it is required to hold. There will then be two kinds of assertions:

- Announcements, where the compiler believes the programmer, and
- Assertions, where the compiler checks the programmer, possibly using the announcements or other means.

8. SUBTYPES

If assertions are to become part of the data types of objects, we are going to have to have convenient ways of adding and removing assertions from a data type. Removing assertions can be left to a coercion (this is similar to using a subtype when a type is required in the DOD's languages), but adding assertions will have to be done by a proof or as an implicit consequence of a run-time test (IF test THEN test true ELSE test false FI).

APPENDIX

Intuitionistic propositional calculus can be built from data types: procedural proofs of Heyting's axioms are listed below. Each entry in the following list is of the form "proposition --- proof".

```
P => (P AND P) ---
   LAMB(P a) STRUCT(P first, second): (a, a);

(P AND Q) => (Q AND P) ---
   LAMB(P AND Q ab) (Q AND P): (second OF ab, first OF ab)

(P => Q) => ((P AND R) => (Q AND R)) ---
   LAMB(P => Q ab) ((P AND R) => (R AND R)):
      LAMB(P AND R ac) (Q AND R) :
         (ab(first OF ac), second OF ac)

((P => Q) AND (Q => R)) => (P => R) ---
   LAMB((P => Q) AND (Q => R) abbc) (P => R) :
      LAMB(P a) R: (second OF abbc)((first OF abbc)(a))

Q => (P => Q) ---
   LAMB(Q b) (P => Q): LAMB(P a) Q: b

(P AND (P => Q)) => Q ---
   LAMB(STRUCT(P first, (P => Q) second) aab) Q :
      (second OF aab)(first OF aab)

P => (P OR Q) ---
   LAMB(P a)UNION(P, Q): a

(P OR Q) => (Q OR P) ---
   LAMB(UNION(P, Q) ab) UNION(Q, P) : ab
      # Algol 68 mode equivalencing does this one ! #

((P => R) AND (Q => R)) => ((P OR Q) => R) ---
   LAMB((P => R) AND (Q => R) abbc) ((P OR Q) => R) :
      LAMB(P OR Q ab) R :
         CASE ab IN
            (P a): (first OF acbc)(a),
            (Q b): (second OF acbc)(b)
         ESAC

NOT P => (P => Q) ---
   LAMB(P => (MODE M)M af)(P => Q) : LAMB(P a)Q : af(a)(Q)

((P => Q) AND (P => NOT Q)) => (NOT P) ---
   LAMB((P => Q) AND (P => NOT Q) abab) (P => FALSE) :
      LAMB(P a) FALSE :
         (second OF abab) (a) (first OF abab(a)).
```

REFERENCES

[1] Heyting A., Intuitionism, North Holland Publishing Company, 1951.
[2] Lindsey, C.H., Modals, Algol Bulletin AB37.4.3, July, 1974.
[3] Boom, H.J., Extended type checking, in New Directions in Algorithmic Languages, 1976, pp. 27-42, IRIA, also separately as Mathematical Centre report IW60/76, Amsterdam.