stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 105/78 DECEMBER

C.H. LINDSEY & H.J. BOOM

A MODULES AND SEPARATE COMPILATION FACILITY FOR ALGOL 68

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68-00, 68A15

ACM-Computing Reviews-categories: 4.12

A Modules and Separate Compilation Facility for ALGOL 68[*)]

by

C.H. Lindsey[**)] & H.J. Boom

ABSTRACT

This report contains both an informal and a formal description of a modules and separate compilation facility for ALGOL 68, and presents various ways in which these features can be implemented. This specification has been released by the IFIP Working Group 2.1 Standing Subcommittee on ALGOL 68 Support, with the authorization of the Working Group.

KEY WORDS & PHRASES: *ALGOL 68, definition modules, separate compilation, language extension, protection.*

---

*) This report will be submitted for publication elsewhere.

**) University of Manchester.

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

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

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

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

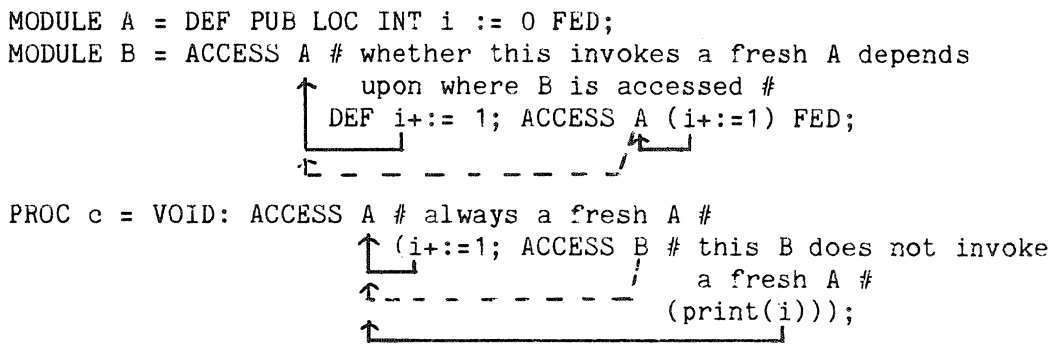
```
MODULE C = ACCESS A, PUB B DEF ... FED
```

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.



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 upcn 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 F00 is accessed as follows:
- The accessor makes a new activator F001.
- The accessor fills in the entry point-environment pair of the definition module F00 being accessed into 'defmod of F001'.
- 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 F00 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 F00 is called, giving it the activator F001 as parameter.
  - F00 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 F00's requirements have been met, F00 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 F00 (or nil if there is none). This copied activator F002 is termed the "principal" activator of F00, 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, F00
    - 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 F00 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 68 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.