stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 77/77 JANUARI

H.J. BOOM

SEPARATE COMPILATION, DEFINITION MODULES, AND BLOCK-STRUCTURED LANGUAGES

Prepublication

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): 68A30

Computing reviews Category: 4.21, 4.22

SEPARATE COMPILATION, DEFINITION MODULES, AND BLOCK-STRUCTURED LANGUAGES *)

by

H.J. Boom.

ABSTRACT

A "definition module" consists of a group of declarations that may be invoked remotely in order to make them available at the point of invocation, much as a procedure can be called remotely. The definition module turns out to be a natural unit for program composition. Using definition modules, it is possible to compile parts of a program separately without loss of security or program structure. This paper discusses how such mechanisms can be installed in a high-level, block-structured, stack-oriented language, and the benefits to be obtained thereby.

KEY WORDS AND PHRASES:
Definition module, module, separate compilation.

---

*) This report will be submitted for publication elsewhere

1. INTRODUCTION

Sometimes, a program gets too large for the logistic support available to construct it as a single unit. The size of the source text reaches limits imposed by the operating system, source listings become inconveniently large, card decks no longer fit in boxes, and large compilation times imply unacceptable turn-around time. At this point, it is necessary to break the program into pieces which can be compiled separately of each other. If the pieces are well-chosen, it will be possible to recompile and alter them relatively independently of one another. This is often true if the division is done according to the modular structure of the program; however, it may sometimes be practical to divide it in other ways, and it may be inconvenient to separately compile a module. It is useless, for example, to try to recompile a macro independently from its calls.

One of the traditional ways of compiling parts of a program independently is to break off single procedures, which are then compiled separately using the same environment as the main program. No global variables are available, except those built into the language definition. The only way of communicating with the procedure is via its parameters. Indeed, calls of the procedure cannot even communicate with each other unless the caller is willing to maintain the communication convention. Some structured programming fans will be very happy about this restriction; others quite unhappy. We gain explicit

environmental control of all our procedures. They can access only that data we explicitly give them, and nothing else. On the other hand, we are forced to pry unpleasantly deeply into the internal structure of the procedure. If the procedure has to maintain a history, the caller has to do it for him, and he must then know the exact form the history takes, in order that he can do it properly. Rather than wisely restricting the ability of the procedure to cause damage to global variables by eliminating them, we have forced every caller to meddle with the internal structure of the procedure. Any fool who comes along to use the procedure, furthermore, can interfere with the procedure via its internal data base, which the fool is forced to deal with. He cannot even leave it alone out of ignorance.

It is clear that, just as with procedures within a block-structured language, our separately compiled procedures must have access to global variables. These are necessary for two reasons,

- to provide a means of maintaining a history, and
- to provide a place to keep a data base which is jointly managed by a group of procedures.

Before we discuss traditional means of accessing such global variables, let us consider linkage. Suppose a program consists of a number of procedures, compiled separately, which nonetheless call each other. Some linkage mechanism must be provided to connect these procedures together. There are two parts to such a linkage mechanism: In the source language within each separately compiled procedure, a programmer must be able to specify which other procedures are to be called, possibly by "names", and when compiling these other procedures, he must then be able to specify their names. We shall call such names "external names". Secondly, there must exist a "linker" which, in some environment associating these external names with the procedures, connects the procedures to each other instead of to their names.

Typical linkers do not consider the procedural structure of a program, but see only

- pieces of code,
- names,
- "external definitions" of names, and
- "external references" to names.

The linker determines some order in which the pieces of code are to be placed together in memory when executed, and this determines the address for each externally defined name. It then takes the address for each name and fills it in at every external reference to the name. In this way, the names are replaced by (relocatable) machine addresses. Conceptually the linker merely accomplishes a change of notation, since machine addresses are just another kind of name.

There is no reason why names should always refer to

procedures; they could just as easily refer to data or to empty storage space. This is used to provide global variables. Languages like Fortran and PL/I have COMMON and EXTERNAL variables. Each COMMON block or EXTERNAL variable has a name. The linker reserves space for the name to refer to and perhaps fills the space with initial data, and uses the external name to grant the procedure access to the storage.

The scheme has serious drawbacks.

First of all, there is no hierarchical structure of program pieces There is simply a vast sea of fragments, each of which proclaims its name and hopes that no other has the same name. The structuring primitives of the programming language are available until one reaches the practical limits of single compilation; after that one is tossed to the waves.

There is no reason to abandon structure just because a problem is large; contrariwise, it is just then that structure becomes indispensible.

Secondly, any procedure may access any datum or any other procedure just by knowing its name. This causes serious security risks, and makes independent proofs of correctness for independent groups of procedures virtually impossible.

2. HOLES

It would be nice if the next step were to suggest itself, but it does not.

Why not extend the block structure of the language to separate compilation?

Experience has shown that block structure is a practical device for structuring programs. It may not be perfect, but it works, and the following remarks apply equally well to a number of other schemes for structuring the name space.

Consider a program. It may consist of a begin and an end, enclosing some sequence of statements, expressions, and declarations, which may in turn contain more statements, expressions, and declarations. We shall treat a procedure declaration as if it contains an expression, called a "routine text", whose value is the procedure being declared. We require that, given sufficient context to determine type conversions, syntactic structure, etc., any statement or expression (henceforth called a "unit") can be cut out and separately compiled (fig. 1, 2). In particular, this includes the routine texts of procedure declarations. No restrictions are placed on nonlocal variables involved in such units except the restrictions inherent in the block (or other) structure of the language. We shall discuss implementation of such a scheme later; first, let

us explore the practical consequences for program structure.

This chopping process need not restrict the kinds of programs acceptible to an implementation; it affects only the manner in which the program text is presented to the compiler. If the language was good, it remains good; if it was bad, it remains bad. It is conceivable that a programmer leaves his program unchanged and alters only the chopping when he is presented with a larger or smaller computer environment.

This chopping process tends to make the block structure clearer. Large programs in existing block-structured languages tend to have unwieldy parenthesis matching. To find the end matching a begin across forty pages of program text is a nontrivial operation, even if the compiler helps by printing nesting-level numbers and reformatting the source text according to its parse tree. By cutting out large chunks and compiling them separately, a block becomes much more readable. It may even be useful to cut a program into pieces and feed all the pieces into the compiler at one time.

Some readers might object that a forty-page block is an atrocity, and that it should already have been cut up by dividing it into procedures and calling them. This objection does not apply, however, when the forty pages themselves consist mainly of pages of procedure declarations. Such declarations may themselves be nested, and may be arranged in the block structure so as to enable certain shared and controlled use of nonlocal indicators. Remember, nearly every procedure call uses a nonlocal identifier to identify the procedure to be called.

Experience shows that large programs in languages with block-structured chopping [2] tend to be written as a large number of pieces, each of which is a few pages long, and may contain some small number of procedure declarations. Many of these pieces contain "holes", which are the sockets into which the other pieces are placed. The pieces form a tree structure, just like that of the block structure they represent.

When a piece of the program is compiled, the compiler reads its source text and an "environment file". The environment file tells the compiler the modes, names, and access algorithms for all global indicators. The compiler produces an object code file and zero or more environment files as output. One environment file is produced for every hole in the source text of the piece being compiled. The environment files can later be used to compile other pieces that fit into the holes, as shown in figure 3.

One effect of this order of compilation is that the compilation of the main program cannot use any information from the stuffing for the hole, although the hole may require information from the main program. This unidirectional flow of information makes simple chopping fit very well into block-

structured languages.

3. LIBRARIES

Let us now consider the problem of program libraries. A typical program library may consist of a number of procedure, mode, operation, and other declarations, which may refer to each other in some way. A fairly natural way to implement such a library is to write a block with a hole:

```
begin
<declaration prelude>;
hole x
<postlude>
end
```

This block is itself compiled in some standard environment. A programmer wishing to use the library must then simply compile his program using the environment file produced from hole "x" when the library was compiled. All the declarations within the <declaration prelude> will be available to him. Furthermore, the <postlude> will be executed after normal termination, allowing the library to close in a neat manner. Unless the program loader and compiler conspire together, though, all of the procedures declared in the prelude will be loaded, whether they are actually used or not.

A programming language usually provides a number of 'standard procedures", which can be called without being declared by the programmer. These can be considered as being declared in a superblock which the compiler places around each program it compiles. We can use the "holes" method of separate compilation to implement the superblock in a pleasing way. It can be written (partially, at least) in the programming language itself, leaving a hole for user programs. It can be compiled by the compiler (once, except if it still needs debugging) to produce an environment for each hole that it contains. One of these environments is selected as the "standard environment"; the others (if any) are simply separately compiled pieces of the standard prelude. The standard prelude is then used for later compilation of programs written by ordinary programmers. It is also used to compile each library, and a programmer wishing to use a library must write his program to fit into a hole left by the library-writer, as shown in figure 4 We call the combination of prelude and postlude around a hole a "circumlude". We select one special hole (if there is more than one), and consider it as the one where the "real" program is inserted; the other holes are for separately compiled pieces of circumlude.

This system looks quite elegant, but it has a serious flaw - it is not possible to use two circumludes together, unless one was compiled within a hole left by the other. This is because it is not possible to have two pieces of code within one hole, or

one piece of code within two holes. The resulting collisions are shown in figure 5. It is possible to compile one circumlude within the other, or the other within the one as shown in figure 6, but this means that at least one of the two circumludes will have to be recompiled by its user - an undesirable situation. The circumludes are thus made to require each other as an artifact of the separate compilation process, even though they may be functionally independent. For these reasons, it is generally agreed by the implementers of the simple chopping method that it does provide a decent answer to the problem of separate compilation of parts of a large program, but that it does not answer the problem of program libraries.

Some system loaders may have useful mechanisms for handling this kind of clash in an elegant way, but we shall seek methods that work with the more conventional plug-in-socket linking loaders. The method will involve two kinds of holes, one for program parts, and one for library definitions. The holes for library definitions must not require external references to the libraries; otherwise, it will be impossible to place more than one library in any given hole.

Another problem with simple chopping is that every complete program will have the same entry point, which resides in the run-time system. Since programs are often referred to using their entry-point names within an operating system, this is inconvenient. Furthermore, there would seem to be no a priori reason to prevent linkage editing of two distinct complete programs (that is, complete from the view of the programming language) into one single linkage-edited object file. Use of identical entry-point names may make this difficult.

4. CLASSICAL DEFINITION MODULES

The classical definition module [1,4] provides a way of separating a group of definitions from the code that uses them. Roughly speaking, a definition module is like a procedure, with one important difference. A procedure may itself contain declarations, and the indicators so declared are available only internally. A definition module usually contains declarations, but the indicators so declared are available to the caller of the definition module as well. For a stack implementation, this means that the local storage claimed by a definition module is not released when the definition module returns control, but is released simultaneously with the local storage of its caller. Let us consider an example.

```
definition d = def
      real x := 0;
      proc y = void : (x +:= i)
      fed;
   int i = 3;
   ...  #(1)#
```

```
begin
    ...
    int i;
    invoke d;
    ...
end
```

The text beginning with "definition d = def" and ending with the matching "fed" is a definition module declaration. No action is performed when, in the normal course of execution, this declaration is encountered, except for that normally performed for procedure declarations. At point (1) in the program, only the indicator "d" is known; "x" and "y" are not known.

Within the block from "begin" to "end", the definition module is invoked. At this time, the definitions it contains are executed, "x" and "y" are created. "x" is initialized to zero, and the identifiers "x" and "y" are made available for use within the block. Even though "i" is redeclared within the block, the definition module itself still refers to the "i (outside the block) that was global to its declaration (just like a procedure). In particular, the procedure "y" uses the "i outside the block, and not the "i" inside the block.

Syntactically, the definition module invocation, "invoke d", is to be considered a declaration.

The idea of a definition module is quite simple, and with a bell and a whistle [1], it is quite surprising how much can be done with it. First, however, we must discuss some implementation questions.

5. INTERACTION WITH BLOCK STRUCTURE

If the programming language is block-structured and its implementation involves a display or static chain for accessing global variables, there is interaction between definition modules and display management. Consider the following program:

```
begin # block A #
int i;
definition d = def # definition module D within A #
      proc p = void: j +:= 1 # procedure E within D #
      real j := 0;
      ... i ...
      fed;

   begin # block B within A #
      int i;
      begin # block C within B #
      invoke d;
      ... i ..
         begin # block E within C #
         call p;
         end;
      ... j ...
      end
    ..
   end
...
end
```

The applied occurrence of "i" in definition module D identifies the declaration in block A, and the applied occurrence of "i" in C identifies the declaration in block B, in accordance with normal block-structure rules. The invocation of D suffices to define the indicators declared within D for use by block C. In particular, the invocation counts as a declaration of "p" and "j". Therefore, the applied occurrence "j" in block C identifies the declaration of "j" in the definition module via the invocation in C.

The display structure reflects these identifications, as we can see in figure 7. We shall use a static chain to represent the display for accessing global variables; copying all display pointers into each activation record would also work but would needlessly clutter the diagram. Block A is entered in an entirely normal manner (fig. 7a-b). The declaration of "d" causes an entry-point-environment pair to be created, just as for a procedure. Blocks B and C are also entered in an entirely normal manner (fig 7 c-d). Then, from block C, definition module D is invoked. An activation record is made for it just as for a procedure (fig. 7e), but when execution of the code within D is complete, the activation record is not popped from the stack. Instead, a pointer to this activation record is placed within the activation record for C (fig. 7f) so that the indicators defined by D can indeed be accessed by C. In effect, C has two display pointers, one pointing to B and one to D. When block E is entered, its static chain pointer points to the activation record of its statically enclosing block, namely c (fig. 7g). Within E the procedure "p" is called. When the definition module D was invoked, it created an entry-point-environment pair for "p", with the environment pointer pointing to D's activation record. "P"

can now correctly access its global variable "j" within the activation record for D (fig. 7h). When control leaves P, and later E, the stack shrinks appropriately (fig. 7i,j). The activation record for D remains in existence until control leaves C (fig. 7k). B and A then terminate (fig. 7l, m).

We shall call abolishing the activation record for a definition module "revoking" the definition module.

On some machines it may be useful to allocate the activation record for D as a component of the one for C. It may even be useful to allocate it before entering C, in effect passing it as a parameter to C. This, in turn, suggests a formal equivalence which will be presented later.

Although we have used entry-point-environment pairs for definition modules above, it should not be assumed that definition modules are normal values with fully general operations defined on them. In particular, there are no definition module variables, and definition modules cannot be passed as parameters. This is to ensure that the identity of the definition module involved in any particular invocation can be statically determined. Otherwise, it will be very difficult indeed for a compiler to determine which indicators should be added to its symbol table at an invocation.

Being able to determine the identity of a definition module statically provides one significant advantage. It may seem to be merely an optimization, but we shall see its true worth later. In the same way as some compilers for Algol 60 implement procedures that are not passed as parameters, one can delay making an entry-point-environment pair for a definition module declaration until the moment that it is invoked. This means that execution of a definition module declaration (but not its invocation) may be a null action. We shall later see that this is crucial for making program libraries convenient.

6. A FORMAL EQUIVALENCE

Let us assume for the moment that the programming language to which we add definition modules permits anything that can be declared to be a parameter. In a language with type declarations, types can then be used as parameters. There are good implementation reasons why this is usually not permitted, but we shall assume it anyway in order to construct a formal equivalence.

For definition modules which are invoked only in deeper ranges than those in which they are declared, we can set up a formal equivalence with procedures. We illustrate this with a definition module that declares four identifiers, "a", "b", "c", and "d", but makes only "a" and "b" available to the invoker. The definition module reads:

```
definition m = def
    public int a := 6;
    public proc int b = int : a+c * (d +:= 1);
    int c := 3, d := 4
    fed;
```

The word "public" is associated with the protection mechanism presented below. It indicates those definitions that are to be made available to the invoker; others are secret. The invoking block reads

```
begin
    invoke m;
    Y
end
```

We can replace the definition module declaration with the following procedure declaration:

```
proc m = (proc(int, proc int) void p) void :

    (int a := 6;
    proc int b = int : a+c * (d +:= 1);
    int c := 3, d := 4;

    p(a,b)
    )
```

We can replace the invoking block with the following call, which passes a revised version of the old block as actual parameter:

```
m(
    (int a  proc int b) void:
      begin Y end
 )
```

This procedure "m" and its call will have the same effect as the original definition module and its invocation.

Having seen this, the question arises why we need definition modules. Are not procedures sufficient?

No.

For valid implementation reasons, most languages place stronger restrictions on parameters than on declarations, contrary to our convention in this section. As a result, not all definition modules can be mimicked in this way. For example, one cannot mimic a definition module which declares a mode or an operator in Algol 68, or one that declares a type in Pascal. The model of definition modules as procedures involves precisely those complications that lead language designers to restrict parameters, such as by requiring data types to be known at compile time. Definition modules themselves, on the other hand,

are well-behaved, and do not hinder sensible implementation; translating them as procedures makes their convenient static properties difficult to discover.

Nonetheless, the translation as procedures demonstrates that definition modules can be implemented with a stack, and it may serve as an implementation model on some systems.

7. DEFINITION MODULES AS CIRCUMLUDES

The formal equivalence suggests treating a definition module as a portable pipe fitting between a hole and its stuffing. The implicit hole in the invocation represents the procedure call generated within the definition module for the formal equivalence. If we consider matters in this way, we may wish to change our syntax for definition modules and invocations.

First, a definition module will contain a "canonical hole", which we shall call a "gap". This will be the gap into which we fit the invoker. In the above example, the gap corresponds to the call "p(a, b)".

```
definition m = def
    public int a := 6;
    public proc int b = int : a+c * (d +:= 1);
    int c := 3, d := 4
    gap
    fed;
```

Second, an invocation is a prefix to a block:

```
using d begin . . end
```

Upon invocation, the definition module is executed; at the gap, the block is executed; and then the rest of the module is executed. If a gap is permitted within a loop, one may repeatedly execute the block (it is not clear that this is a good idea. One might well wish to keep gaps out of loops, conditional clauses, more deeply nested definition modules, etc.).

Only one gap is permitted in each definition module.

With this formulation, it is possible to have declarations after the gap (though what use they would be I can not see). It is also possible to have label definitions, and a postlude to be executed after normal termination of the block. If the program uses jumps, it may be impossible to rely on execution of the postlude.

It is indeed possible to use the gap mechanism to partially replace the hole mechanism in simple chopping. Instead of letting the circumlude call the stuffing for one of its holes, we let the stuffing invoke the definition module containing its gap. The use of a definition module instead of a superblock with

a hole solves the entry-point-name problem mentioned earlier.

8.THE WHISTLE: PROTECTION

One might very well imagine that the writer of a definition module may wish to make some declarations for internal use, and other declarations for external use. The internal declarations should not be accessible to the invoker of the module.

To this end, we specify that one can prefix the word "public" to each declaration in a definition module, even to an invocation of another definition module. If "public" is prefixed, the indicators declared by the declaration will be available to the invoker; otherwise, they are known only inside the definition module. Another scheme is to provide a definition module with a header or footer which lists all the definitions it makes public (and perhaps also all the indicators it inherits as global variables). This has the advantage of providing default protection without requiring internal changes in the code when a chunk is cut out of a program and placed into a definition module. On the other hand, if the definition module consists of essentially trivial declarations of many indicators, the header method effectively requires the definition module to be coded twice. We shall later see that the notation used has implications for the recompilation problem.

The "own" concept of Algol 60 was essentially a kludge to provide something analogous to secrecy in definition modules.

There are some good reasons for choosing secrecy as default instead of publicity. The most important one is that it prevents accidental publication. It is impossible to forget to write "secret" if one must instead write "public" elsewhere.

9. THE BELL: SHARING

With the definition proposal so far, if two invocations of some definition module are executed, two activation records are created. This is usually either silly or dangerous.

Suppose that one wishes to perform structural analysis of some prospective bridge. One may well wish to use standard structural analysis programs instead of developing one's own. One's analysis program might well begin with two definition module invocations

```
invoke stresses;
invoke vibrations;
```

in order to make both standard packages available. A user of a package should not have to be aware of the internal structure of the package. Stresses may well internally invoke some matrix

package (which we shall call matrix), and so might vibrations. It would be silly to have two independent activation records for the matrix package around. Some trick must be found for eliminating the extra one. We must find some way whereby stresses and vibrations may jointly invoke the matrix definition module.

It can also be necessary for program correctness, instead of merely for efficiency, that a definition module be invoked only once.

We invent the "shared" definition module. An invocation of a shared definition module causes a new invocation of a module only if one does not already exist. If one does exist, it makes the old one available again. Since definition modules are invoked and revoked in synchronization with the runtime stack, this does not cause scope problems.

If one wishes to let definition modules be shared between several invocations, the question arises as to the proper point at which sharing occurs. There will have to be some compile- or run-time data structure which records when a definition module has been invoked, so that it will later be known whether it is available for reuse. This "sharing point" is similar to a semaphore, except that it provides sharing instead of exclusion. There are some multiple invocations of a definition module, however, that must clearly be distinct; In a multiprogramming system, for example, each separate job invoking a definition module will usually want its own separate invocation to prevent unwanted interference between users. The operating system is thus a conceptually unwise place to place this sharing point. Furthermore, each definition module may require an environment for proper execution. Different executions of the block in which it is declared will provide it with different environments, which must be distinguished at invocation. The sharing point must therefore not be more global than the point at which (conceptually, at least) the definition module declaration is executed and its entry-point-environment pair is constructed.

On the other hand, the sharing point will have to be more global than the first invocation of the module; otherwise it will have no way of determining that it is indeed the first. We must conclude that execution of the definition module declaration must construct the sharing point, just as it constructs an entry-point-environment pair. The proper sharing point for an invocation is therefore found when the definition module declaration is identified. If a sharing point is actually built at run-time, the definition module declaration will require execution, and, as mentioned before, this is incompatible with a convenient library mechanism. Sharing must therefore be decided at compile time and be statically determinable. We adopt the convention that, if another invocation of a shared definition module exists in a range which (statically) includes the current one, then a new invocation will simply access the outer one once again.

Shared definition modules become important when definition modules invoke each other. A shared invocation is then a "requirement" by some definition module D that some other definition module E be made available to it. If there is an invocation of E active at the point of invocation of D, that invocation's activation record is given to D as parameter (the compiler must arrange this behind the scenes; the programmer need not concern himself with the mechanism). Otherwise, the compiler secretly invokes E, making its indicators available to D (but not to the range invoking D, unless this range itself contains invocations of E.). Other invocations of E will of course share this new activation record. It is important to realize that in this version of sharing it is statically determinable when definition modules are invoked.

A typical large program or program library will consist of a number of definition modules, most of which are compiled in the same environment. These definition modules may use the sharing/requiring mechanism to require that other definitions modules are invoked. For each definition module, the compiler determines which others are required directly or indirectly by it and sees to it that the others are invoked as required.

If a definition module is used to define a data structure, one may be tempted to use multiple invocations of the module to construct multiple versions of the data structure, each with its own private variables and administration. The definition module is then treated as a data type definition, and each invocation creates a value of this type. It seems preferable to have a proper data structuring facility in the language, instead of misusing definition modules. Attempts to adapt definition modules to this misuse leads to non-stack-oriented features much resembling Simula classes. While classes are certainly not to be despised, they do not constitute the efficient modularity and separate-compilation facility discussed here. It gains its efficiency from a relatively normal stack implementation, which makes it difficult to use it like Simula classes to build data structures. It is often possible to pervert one feature into doing the job of another, but it cannot be recommended. The proper way to construct multiple versions of a data structure is not to use multiple invocations, but to declare a procedure which constructs versions. The full data structuring mechanism of the programming language can then be used for manipulating the defined structures, and the procedure declaration may of course be placed in a definition module.

It may be that unshared definition modules are not needed in practice. Nonetheless, the whole nature of procedure calling needs further study, since procedures, definition modules, classes, coroutines, and parallel proccesses appear to have much in common that is still poorly understood.

10. INTERACTION OF THE BELL AND THE WHISTLE

What does it mean to have a shared secret invocation of some module m within another module n?

It means that:

- (secrecy) the indicators defined by the invocation of m are not published by n outside its own invocation.
- (sharing) other shared invocations of m will receive the same activation record, under the usual conditions for nonsecret invocations. Secrecy does not interfere with sharing.

A typical large program will consist of a main block and a number of definition modules. Each of these is compiled separately. Each definition module and the closed clause may "require" other definition modules. Normally, the sharing mechanism will suffice to ensure that each definition module will be invoked only once.

Definition modules may be needed within other definition modules, blocks, or procedures if the standard environment of a program is implemented by compiling it as a superblock, with the users' programs compiled into a hole within the superblock. The user's definition modules may thus end up within constructs in the superblock.

11. SEPARATE COMPILATION OF DEFINITION MODULES

Each definition module must be declared within some environment. This environment is necessary for the definition of its nonlocal indicators. Each invocation, on the other hand, needs to know which indicators are declared by the definition module, and which other modules are required by it. If some method is found for communicating these data, definition modules can be compiled separately.

To enable such communication, the compiler recognises "environment publishers" in source code. An environment publisher is a construction that indicates that the compiler must place environment information on an output file.

When a definition module is compiled separately, the compiler reads a file of previously published environment information in order to be able to compile correct code for global indicators. The environment information must therefore contain the declared indicators, their modes, and their access algorithms. The compiler will produce, in addition to object code, a "definition file" of analogous information for the definitions and requirements in the definition module. This definition file will be included in the compiler input when a program is subsequently compiled which invokes or requires the definition module. The environment of the invoking program must

contain the environment of the definition module as a proper or improper subset, otherwise it will be impossible to guarantee the definition module its global indicators.

Most definition modules will probably be compiled in the standard environment produced by compiling the standard prelude.

As presented here, a definition module declaration does not require execution (but there are variations on the sharing mechanism which do require it). This is crucial to the possibility of a convenient program library facility. Otherwise, at a hole or env provided for the writing of libraries, an unknown number of separately compiled definition module declarations must be executed. Normal linkage editors provide no help in accomplishing this.

An implementation may, of course, construct the environment information from a hole in such a manner that it can be used for compiling a definition module. The crucial difference is that there must be one and only one stuffing for a hole in any linkage-edited object code; whereas there may be zero, one, or many definition modules for each environment. Since a definition module declaration requires no execution, no external reference need be made by the object code of an environment publisher, and there is no need to choose the correct definition module declaration to insert. That is determined at invocation, not at declaration. Figure 8 shows how a main program, a definition module, and a program that uses the definition module can all be compiled.

12. THE COMPATIBILITY CHECK

Some check is needed to ensure that at load time the object code loaded is indeed that corresponding to matching environments. One way of doing this is by a serial number. Each environment or definition file is furnished with a serial number, different from that of each other environment. Object code is furnished with the serial number of the environment in which it is compiled, the serial numbers of any environments or definition files it may contain, and the serial numbers of the definition modules it invokes. These serial numbers are compared for identity at either linkage editing or execution time. If the linkage editor accepts sufficiently long names, the serial number can simply be appended to the external names already involved in linkage.

Every linkage editor should accept really long names. Few do. The limit is usually between six and eight characters, which is absurdly inadequate even for just the identifiers commonly used in modern programming languages. Here we have another example of misplaced efficiency considerations. The same stupidity often occurs in job control languages. In order to save microseconds per job the syntax is contorted to

incomprehensibility. It might be understandable if the job control analyser were usually called from the inner loop of a matrix inversion or multidimensional integration, but I have so far seen no evidence of this. More time is probably wasted by incomprehensible syntactic errors in the job control statements than can possibly be saved by the syntax. Furthermore, the job control language and linking loader are parts of the system that every user uses fleetingly for every job, and for every programming system: they must therefore be general.

Because linkage editors are uncooperative, the serial number check must usually be performed at run time. A complete interface specification would make a fine serial number for a linkage editor, but at load time it will usually require unconscionably much storage. One is therefore required to build an arbitrary unique name generator. Concatenating the machine serial number and the date and time is usually sufficient to construct a unique name, provided that a global semaphore is tripped sufficiently long to prevent another job from creating the same name at the same time. Generation of unique names should perhaps be considered an operating system primitive, or be done according to standard system conventions. One mechanism might be to let every program library contain a unique name counter, which can be used by the compiler when it places object code into the library. Unfortunately, this requires that some form of name scope be recognized when combining libraries, since different libraries will have independent name counters.

13. RECOMPILATION

After a definition module has been compiled, placed in a library, and used by many customers, it may become necessary to recompile it. This may be to fix a bug, to improve performance, or simply to reconstruct definition or object-code files after they have been inadvertently destroyed. If the definition module is recompiled, the access information or serial number in the new definition file may not be identical to that in the old, even if the same indicators with the same modes are published before and after.

Some mechanism should exist to obviate recompilation of all programs using the definition module. Two mechanisms can be used. First, the recompilation can differ from the original compilation in that the compiler is provided with the old environment file and required to create object code to match. Second, the access information can be computed in such a way that it depends only on the published indicators and their modes (and perhaps on their order).

If the entire access interface is accepted and checked by the linkage editor, either method is sufficient. If, instead, an arbitrary serial number is used for the check, the compiler must

receive the old environment so that it can compare it with the new and avoid generating a new serial number. This is necessary no matter whether the access information depends on anything other than the published indicators or their modes.

If the environment file contains information for performing the constant propagation optimization (for a language providing manifest constants, it might well do so), constraints on recompilation will be more severe than if it contains no such information.

If each definition module has an explicit interface specification, as discussed earlier under "secrecy", it can be used to determine the run-time interface independently of the rest of the content of the definition module. If, as discussed under "sharing", the invoker of a definition module must invoke the other definition modules it "requires", then the identities of the other definition modules must also be part of the interface. (The only alternative to this appears to be to search tables at run time to find shared activation records of definition modules. This is tolerable if definition modules are only rarely invoked. If they are used only to link together the large-scale structure of the program, this will probably be the case; if they are used in other ways, it may not be.

14. EXAMPLE: PLOTTING SOFTWARE

A program library of the traditional sort consists of a number of procedures. Quite often these procedures attempt to communicate behind the scenes via COMMON storage or some other mechanism. A set of routines for managing a graph plotter, for example, might consist of a number of routines:

```
plinit  to initialize the plotting system.
move    to move the pen to a point.
down    to lower the pen.
up      to raise the pen.
letter  to write text on the picture.
axis    to plot axes.
newplot to start a new independent picture.
plend   to terminate plotting and flush any internal buffers.
plwait      (in case of an online plotter) to drain any
        internal buffers in order establish synchronization
        between the program and a human plotter operator.
etc.
```

Such a package of routines usually maintains joint data, such as scaling factors, coordinate transformations, the current pen position, and whether the pen is up or down. If the hardware plotter interface is via a device that encourages data blocking (such as magnetic tape, or a packet switched transmission line) or if buffering is required to asynchronize plotting and computing, the plotting routines may well wish to maintain

internal buffers. All of this internal information must not be directly accessible to the user of the plotting system, lest he damage its integrity.

In Fortran, a typical one of these plotting routines might well begin

```
SUBROUTINE AXIS (...)
COMMON /SECRET/ POSX, POSY, AXIS, AYIS, ...
...
```

This leaves the routines open for mutilation by anyone who cares to place a COMMON /SECRET/ statement in a program. This is better than the situation in normal Algol 60 implementations. There, independently compiled procedures are usually denied the right of communicating via COMMON storage or anything similar. This leaves two alternatives. First, one can escape to another language to write the plotting procedures (or just the communication interface). This is an admission of inadequacy of the separate compilation mechanism, but is tolerable in practice. Second, one can demand that all of these secret variables be explicitly provided as extra call-by-name parameters each time that a plotter procedure is called. This makes the secret variables excessively prominent and actively encourages tampering; we can not consider it a solution.

If definition modules were used to place these plotting procedures in a library, the situation would be different. The library could be compiled as a single definition module:

```
def plotting = def

    proc move =  ...      ,
    proc down =  ...      ,
    proc up   =  ...      ,

    ...

    secret real posx, posy, axis, ayis, ... ;

    c the body of "plinit" c;

    def;
```

A program wishing to use the plotting system would invoke it:

```
invoke plotting.
```

It would be impossible to call a plotting routine without first invoking the definition module; this guarantees that the plotting system will be properly initialized if used. Furthermore, the routines can communicate via the secret variable, without interference from the user.

The plotting definition module will have been compiled beforehand (presumably by some system programmer) using the standard environment of the programming language (In Algol 68, this would provide it all the definitions of the standard prelude.). Since the user program will also be compiled in this environment (or a deeper one), it will be compatible, and can thus invoke the definition module.

There is one possible difficulty. This mechanism appears to imply that the object code for all the procedures in the plotting module will be loaded when any of them is required: they are all part of one module. This can be avoided if the compiler follows the sensible practice of having the object code for procedures loaded only if they are used other than in their declarations.

15. OPEN PROBLEMS

A number of problems can be seen. This paper has hinted at some partial solutions, but they should not by any means be considered final.

- It is necessary to have a library search for required modules. These modules must be invoked at a proper block nesting level, which must somehow be determined. It is probably a mistake to gather all such implicit invocations in some outer block; they should be done where required, even repeatedly, unless other invocations exist. It is not sensible to invoke a graph plotting definition module at the operation system level because two different jobs wish to use it on the same day, because then they may get their plots tangled. (It might be sensible to load the code once if it is reentrant, but that is a different matter.)

- It is necessary to be able to recompile a definition module (perhaps to fix a bug) without having to recompile all the programs that invoke it. Recompilation is necessary to match an old interface.

- It is not clear what the lifetime for a definition module invocation should be in a stackless or blockless language.

- There is the problem of side effects of definition modules. This will probably have to be handled like the side effects of procedures - to be avoided but impossible to police. And who is to judge effects and say which of them is unintended, and therefore the side effect?

- More explicit control is needed by the invoking program over name visibility within the invoked modules. This is not as important as a security measure to imprison naughty definitions, but as a means to provide an overview of name and definition flow.

- The invoking or requiring program and the invoked or required definition module require some means of establishing an interface. Many methods can be suggested. At one extreme we have complete duplicate specification, with a load-time, link-time, or run-time check. At the other extreme we have a single specification from one side. Compilation produces an environment-specifying file which is swallowed again by the compiler when it compiles the program at the other side of the interface. Each of these methods has its advantages and disadvantages. It is clear that some flexible combination of these methods is required in a convenient system, or else a new method altogether.

ACKNOWLEDGEMENTS

The author wishes to thank the members and observers of IFIP working groups 2.1 and 2.4 for much valuable discussion and criticism of an early draft of this paper.

REFERENCES

[1] Steve Schuman, Toward modular programming in high-level languages, Algol Bulletin 37.4.1, July 1974, 12-23.
[2] S. R. Bourne, A. D. Birrel, and I. Walker, Algol 68 C Reference Manual.
[3] I. F. Currie, Modular Programming in Algol 68, Algol Bulletin 39.4.1, February 1976, 13-19.
[4] Charles Lindsey, A proposal for a modules facility in Algol 68, Algol Bulletin 39.4.2, February 1976, 20-29.

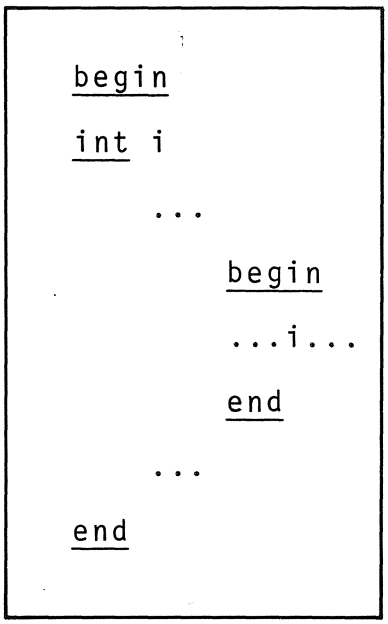

Fig. 1. A program.

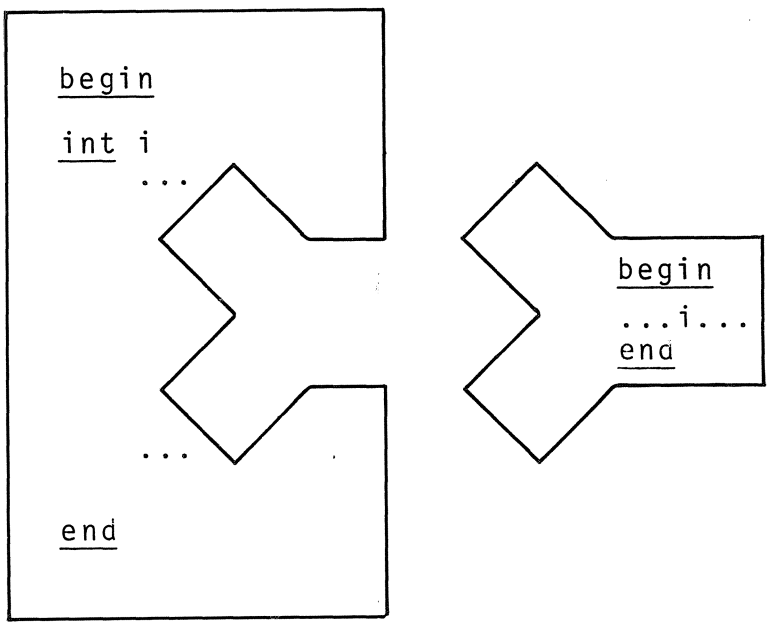

Fig. 2. A program cut up

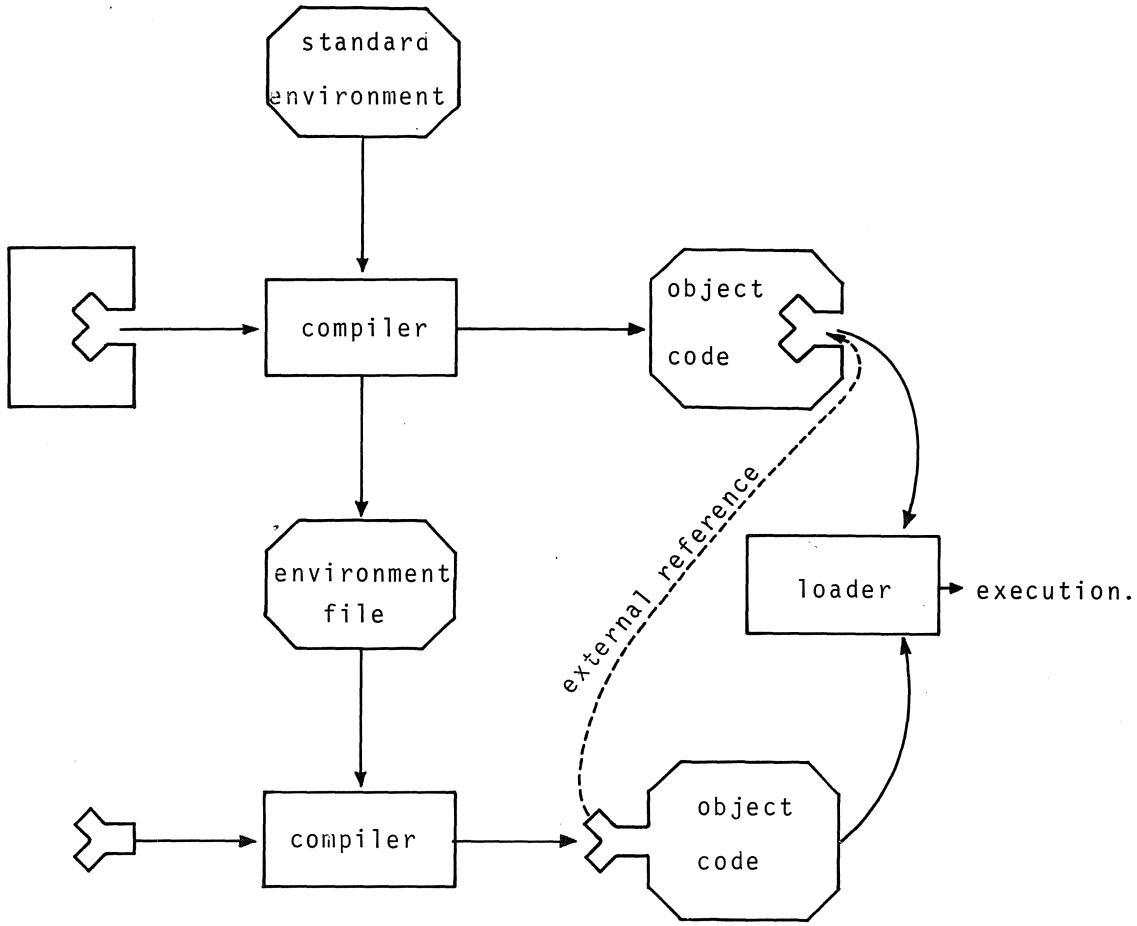

Fig. 3. Separate compilation with a hole.

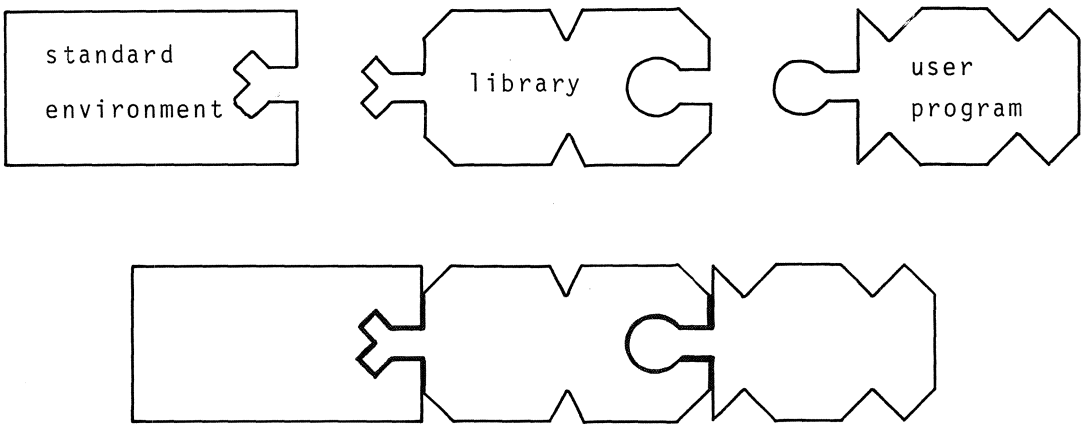

Fig. 4. Using a library

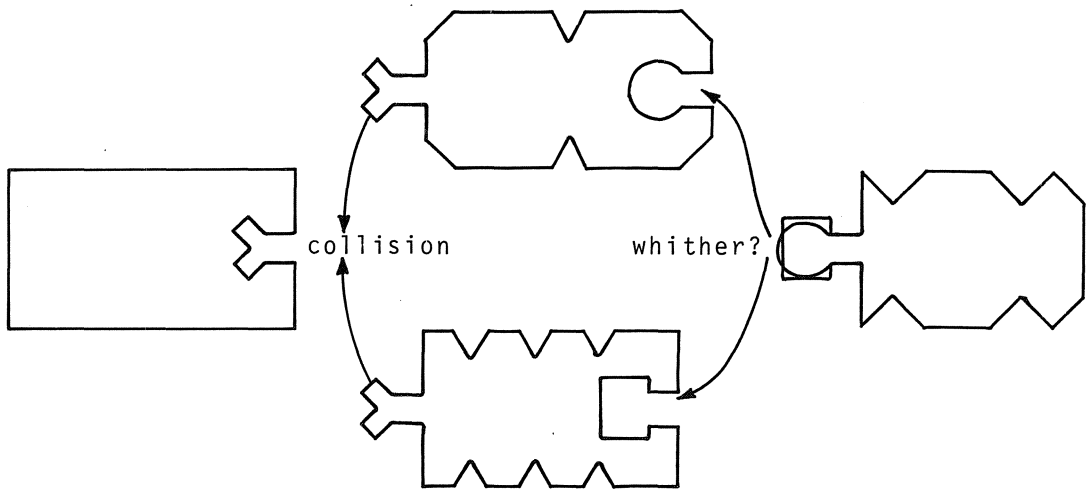

Fig. 5. Two libraries don't work.

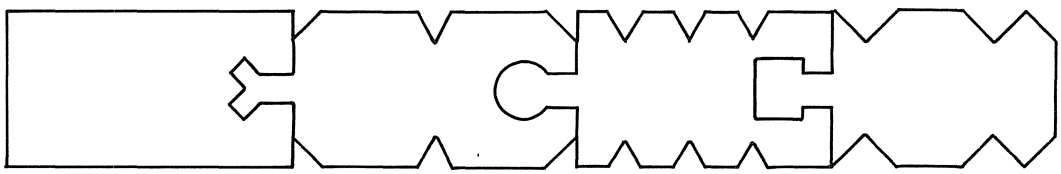

Fig. 6. An unsatisfactory technique.

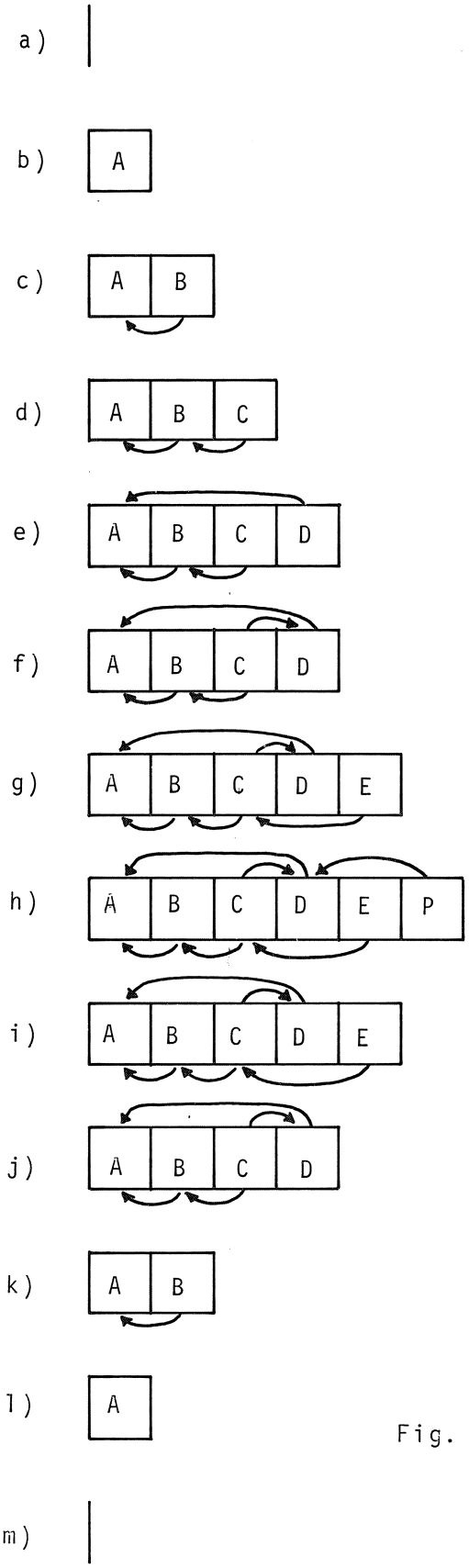

Fig. 7. Static links.

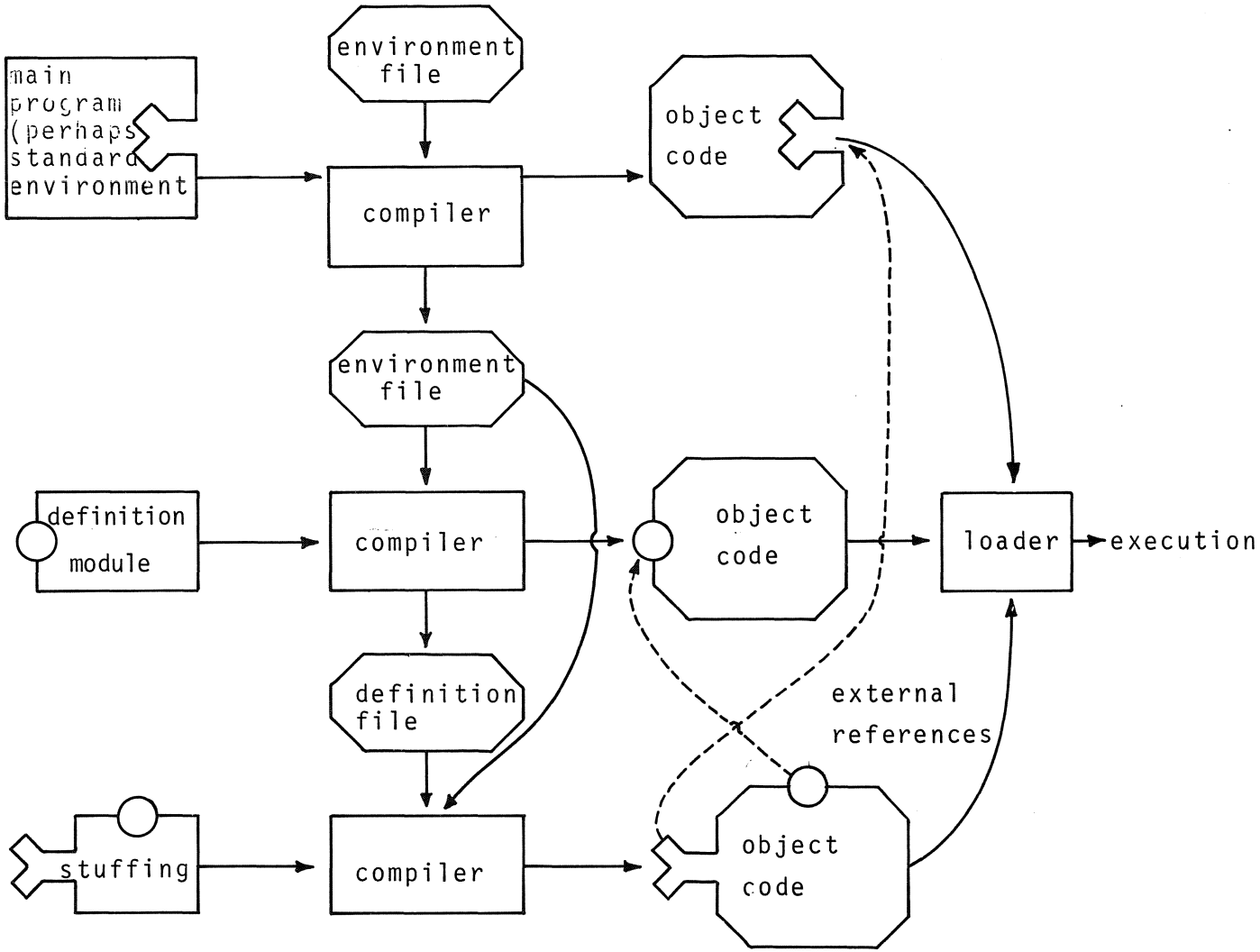

Fig. 8. Separate compilation with a definition module.