15379 NATIONAL LIBRARY
OTTAWA

BIBLIOTHÈQUE NATIONALE
OTTAWA

NAME OF AUTHOR....Samuel John Wilmott....

TITLE OF THESIS....On the Ordering of Modes in ALGOL 68

....

UNIVERSITY....University of Alberta..

DEGREE FOR WHICH THESIS WAS PRESENTED....M.Sc.....

YEAR THIS DEGREE GRANTED....1973.....

Permission is hereby granted to THE NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed)....Sam Wilmott

PERMANENT ADDRESS:

30, Mark Lane,
R.R. 7, Victoria,
British Colombia.

DATED....June 18....1973

NL-91 (10-68)

THE UNIVERSITY OF ALBERTA

ON THE ORDERING OF MODES IN ALGOL 68

BY

(C) SAMUEL JOHN WILMOTT

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

AND RESEARCH IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

SPRING, 1973

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommended to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled ON THE ORDERING OF MODES IN ALGOL 68 submitted by Samuel John Wilmott in partial fulfillment of the requirements for the degree of Master of Science.

Barry J. Mailloux
(Supervisor)

T.A. Marsland

Roman A. Maciuk

DATE May 3, 1973

ABSTRACT

This thesis discusses the problems caused by united modes in ALGOL 68. An algorithm is presented which uniquely orders modes. Applications of the algorithm are described which eliminate the difficulties associated with united modes. It is also shown how mode ordering may be used to speed up and increase the power of an ALGOL 68 implementation.

ACKNOWLEDGEMENTS

I wish to thank Professor B.J. Mailloux, my supervisor, for his advice and guidance throughout my studies and the preparation of this thesis. The help and encouragement offered by my fellow students, especially L.K. Thomas, is gratefully acknowledged.

The financial support received from the National Research Council of Canada, in the form of grants, is appreciated.

TABLE OF CONTENTS

CHAPTER I

Introduction

The Report on the Algorithmic Language ALGOL 68 [1] (hereinafter referred to as the Report) defines a new programming language intended to be the official successor to ALGOL 60 [2]. ALGOL 68 is designed around the concept that all program and data structures, desired of a computer, have convenient representation in a general purpose programming language. In such a language the compiler can reasonably restrict the use of any structure. This restriction has two consequences: the compiler can detect a large class of erroneous program constructs that would otherwise be treated as valid, and the limitation on the use of any structure allows the compiler to deduce much more exactly the intention of any program construct. The latter facility both enables a program to be much less explicit about the operations it requires to be performed, and enables the compiler to produce more efficient object programs based on its increased knowledge of what is intended by those programs.

The format and intended use of a data object is

described by its mode. Besides providing primitive data types and mechanisms for constructing complex data objects, the Report provides for data objects whose mode is either not known exactly at the time a program is compiled or whose mode may vary when the program is executed. These data objects have united modes, and have at any time during the execution one of a number of moods, or non-united modes, specified by the united mode. In Section 3.3, comparisons will be made between the representation of objects of united mode suggested by previous authors, and a more efficient representation introduced in that section.

The thesis presupposes a certain familiarity with ALGOL 68. Introductory expositions of the language are contained in [7, 8]. The first occurrence of any technical term or notion from the Report will, where necessary, be followed by a reference to the particular section of the Report in which the definition may be found. [R1.1.6i] will, for instance, denote Section 1.1.6.i of the Report. The concepts discussed in the thesis are illustrated by a number of short examples in Chapter III, and a longer one in APPENDIX B.

Chapter II of the thesis presents an algorithm which generates a unique ordering for any set of modes in ALGOL 68. Chapter III describes the uses of the algorithm in inplementing ALGOL 68. The implementation of the algorithm itself is discussed in Chapter IV. Finally Chapter V summarises the advantages gained from use of the algorithm.

CHAPTER II

The Algorithm

2.1 Outline of the Algorithm

A mode[R2.2.4.1] is any terminal production of the metanotion[R1.1.3b] MODE[R1.2.1a]. Two modes are equivalent if either they are the same sequence of small syntactic marks[R1.1.2a], or there is some sequence of production from the rules 7.1.1dd to 7.1.1jj in the Report which will derive one of the modes from the other.

Given an unordered set of modes, the algorithm will produce an ordered set of classes of these modes such that modes within a class will be equivalent. The ordering so imposed on any given pair of modes in the set will be independent of both the initial representations of the modes and the other modes in the set being ordered. The relation defined between modes in each class is reflexive, symmetric and transitive, and is therefore an equivalence relation. The relation between non-equivalent modes is transitive, and is therefore an ordering. Also each pair of modes has a mutual ordering, so that any given set of nonequivalent modes has a unique linear ordering.

The algorithm is iterative. At each stage of the algorithm, a class is selected for sub-division. The selected class must contain non-equivalent members. If there is no class which can be so selected, the algorithm is terminated.

On sub-division of a class, its parts are assigned positions in the ordering of classes being established. An attempt is then made to select another class for division. At the first stage of the algorithm, there is just one class, containing all the modes under consideration.

The algorithm is based on that of Zosel [9] and only differs from it by assigning the parts of a sub-divided class their position in the ordering being established. The function of ordering is therefore added to that of equivalencing.

The selection of a unique class to divide, and a unique method of division, forms the crux of the algorithm. Section 2.2 describes the algorithm in detail, and section 2.3 discusses the correctness of the algorithm, based on the above-mentioned uniqueness.

2.2 Statement of the Algorithm

The algorithm is most easily described as ordering the modes occurring in a given particular-program[R2.1d].

Consider all declarers[R7.1.1a] in the given particular-program. Protect them[R7.1.2b] and continue to develop them[R7.1.2c] as follows. Represent each declarer, which is not a void-declarer[R7.1.1z], by a unique designation Mi, so that M0, M1, M2, ..., Mn (for some integer n) represent every declarer in the particular program. In the following, "the mode represented by the declarer designated by Mi" is referred to more simply as "the mode Mi". Replace each Mi, which represents a declarer which is a mode-indication[R4.2.1b], by that Mj which represents the actual-MODE-declarer[R7.1.1b] of the indication-defining occurrence[R2.2.2c] of the mode-indication. Repeat this replacement until there are no further replacements to make. This process will fail to terminate only in the case for which the declaration[R4.4.4c]-condition has been violated. It can always be determined whether the process will fail to terminate, because in such a case, and only in such a case, will any mode-indication be successively replaced more times than there are mode-indications in the complete set of modes being ordered [10]. The algorithm can be forced to terminate when such a condition is detected.

The Mi which ultimately replaces the representation of a declarer is used to represent the mode of the declarer. Let the constituent[R1.1.6f] modes of Mi be those Mj which are the modes of the declarers that are descendents[R1.1.6e,f] of the declarer represented by Mi but

are descendents of no declarers which are themselves descendents of the declarers represented by the Mj.

Each mode Mi is represented by a terminal (the T(Mi) below), a possible sequence of tags, and zero or more constituent modes, represented by other Mj. This notation is due to Peck [10].

For each mode Mi define T(Mi) as follows.

(a) If the mode of the declarer represented by Mi begins with 'LONGSETY integral', 'LONGSETY real', 'boolean', 'character', 'format', 'union of', 'row of', 'reference to' or 'structured with', let T(Mi) be that sequence of symbols.

(b) If the mode of the declarer represented by Mi begins with 'procedure' and ends with a void-declarer, T(Mi)='proc void'.

(c) If the mode of the declarer represented by Mi begins with 'procedure' and ends with other than a void-declarer, T(Mi)='procedure'.

CONST is used to select constituents of modes. Each Mi has a number of constituent modes. Let N(Mi) be this number. Define, for $1 \leq n \leq N(Mi)$, CONST(Mi,n) to be the nth constituent mode of Mi in the textual order of the declarer represented by Mi.

For each Mi for which T(Mi)='structured with' define TAG(Mi,n), for $1 \leq n \leq N(Mi)$, to be the tag associated with

CONST(Mi,n).

For each Mi such that T(Mi) is not 'union of', define L(Mi)=N(Mi). For united modes, L(Mi) is defined in Step 2.

The algorithm follows.

Step 1. The Mi are initially ordered based on the values of T(Mi), TAG(Mi,k), and L(Mi), as follows. First define

order[1] = 'structured with'

order[2] = 'union of'

order[3] = 'boolean'

order[4] = 'character'

order[5] = 'format'

order[6] = 'reference to'

order[7] = 'row of'

order[8] = 'procvoid'

order[9] = 'procedure'

order[10] = 'integral'

order[11] = 'real'

order[n] = 'long' order[n-2] for n>11.

This latter recurrence relationship provides for ordering all INTREAL[R1.2.1e] modes on their corresponding T(Mi). Define, for each pair of modes Mi, Mj under consideration, T(Mi)<T(Mj) if T(Mi)=order[i'], T(Mj)=order[j'] and i'<j'.

Define TAG(Mi,n)<TAG(Mj,m) if the relation is true with respect to some fixed alphanumeric ordering.

Define C(Mi) for each Mi in the set of modes under consideration in the following substeps:- initially let P=0.

(a) Call any Mi an "unassigned structure" if C(Mi) has not been defined and for which T(Mi)='structured with'. Find an unassigned structure Mi such that for all unassigned structures Mine|Mj either

(i) $N(Mi)<N(m(j))$; or

(ii) $N(Mi)=N(Mj)$, and there is some n, $1\leq n\leq N(Mi)$, such that $TAG(Mi,n)<TAG(Mj,n)$, and for all k, $1\leq k<n$, $TAG(Mi,k)=TAG(Mj,k)$; or

(iii) $N(Mi)=N(Mj)$, and for all k, $1\leq k\leq N(Mi)$, $TAG(Mi,k)=TAG(Mj,k)$.

Set C(Mi)=P. Likewise, for all Mj such that $N(Mi)=N(Mj)$ and $TAG(Mi,k)=TAG(Mj,k)$ for all $1\leq k\leq n$, set C(Mi)=P. Set P to be one greater than C(Mi) and return to substep (a).

If no Mi exists which satisfies substep (a), continue to substep (b).

(b) For all Mi for which T(Mi)='union of' set C(Mi)=P. If there is any such Mi, set P to be one greater than C(Mi).

(c) Find an Mi for which C(Mi) has not been defined, such that there is no Mj for which $T(Mi)>T(Mj)$, and such that there is no Mj with $T(Mi)=T(Mj)$ and $N(Mi)>N(Mj)$. Set C(Mi)=P. For all Mj for which $T(Mi)=T(Mj)$ and $N(Mi)=N(Mj)$ set C(Mj)=P. Set P to be one greater than C(Mi). If any Mi exists for which C(Mi) has not been assigned a value, return to substep (c).

If C(Mi) has been defined for all modes in the set under consideration, step 1 is finished and P is the number of classes of modes at this stage in the algorithm.

If C(Mi)<C(Mj) then it can be said that Mi<Mj. The values of C(Mi) partition the set of modes into classes, which will be further partitioned by steps 3, 4 and 5 until each class contains only equivalent modes.

Step 2 determines the distinct classes into which the constituents of each united mode fall, and orders these classes. Sets of ordered classes of constituents are used to order the united modes in steps 4 and 5 in much the same way that ordered contituents are used to order non-united modes.

Step 2. For each Mi such that T(Mi)='union of', perform the following substeps.

(a) Let g=1, h=-1.

(b) Find the smallest value of C(CONST(Mi,n)), $1 \leq n \leq N(Mi)$, such that C(CONST(Mi,n))>h. Define S(Mi,g)=C(CONST(Mi,n)).

(c) Let h=S(Mi,g). If $h \geq C(CONST(Mi,n))$ for all $1 \leq n \leq N(Mi)$, go to substep (d). Otherwise let g be incremented by one and return to substep (b).

(d) Define L(Mi)=g.

Step 3 orders non-united modes on their constituent modes. Constituent modes which have not yet been found to be nonequivalent are skipped over until a mode is found on

which ordering can be done. The skipping is performed independently of the ordering the modes may ultimately acquire. The validity of doing this and the similar mechanisms of steps 4 and 5 will be discussed in Section 2.3.

Step 3. Consider the class of modes for which the following are true.

(a) T(Mi)≠'union of' for any Mi in this class.

(b) There is an n such that $1 \leq n \leq L(Mi)$ for the Mi in the class, and for all Mi, Mj in the class,for all m, $1 \leq m < n$, C(CONST(Mi,m))=C(CONST(Mj,m)).

(c) For the n of (b), there are Mi and Mj in the class such that C(CONST(Mi,n))≠C(CONST(Mj,n)).

(d) For any Mi in the class, and any Mj in another class satisfying (a), (b) and (c), $C(Mi) < C(Mj)$.

If there is no class satisfying the above, proceed to step 4. Otherwise partition the class as follows.

(a) For each Mi for which $C(Mi) > C(Mj)$ for any Mj in the considered class, redefine C(Mi) to be one greater than it was before this elaboration of step 3 was commenced.

(b) Find the value of C(CONST(Mi,n)) which is minimal for the above n and every Mi in the considered class. Redefine C(Mi) to be one greater than it was before this execution of step 3 was commenced for each Mi such that C(CONST(Mi,n)) is greater than the found minimum. Some such Mi exists by condition (c) above.

Return to step 2.

Step 4 separates unions using the L function. This was done for non-united modes in step 1. At each stage of the algorithm, the L function for a mode is the number of its constituent modes that have already been found to be distinct. At the termination of the algorithm, the L function will be the actual number of constituents of the mode. A mode has.

Step 4. Consider the class of modes for which the following are true:

(a) T(Mi)='union of' for any Mi in this class.

(b) There is an n such that for some Mi in the class, L(Mi)=n and for some other Mj in the class, L(Mj)>n.

(c) For all Mi, Mj in the class, for all m, $1 \leq m \leq n$ for the above n, S(Mi,m)=S(Mj,m).

(d) For any Mi in the class, and any Mj in another class satisfying (a), (b) and (c), C(Mi)<C(Mj).

If there is no class satisfying the above, proceed to step 5. Otherwise partition the class as follows.

(a) For each Mi fcr which C(Mi)>C(Mj) for any Mj in the considered class, redefine C(Mi) to be one greater than it was before this elaboration of step 4 was commenced.

(b) For each Mi in the considered class for which L(Mi)>n, for the above n, redefine C(Mi) to be one greater than it was before this elaboration of step 4 was commenced.

Return to step 2.

Step 4 guarantees that before step 5 is begun, L(Mi) for each Mi in a class of unions is the same. Step 5 orders unions on their constituent modes much as step 3 did for non-united modes.

Step 5. Consider the class of modes for which the following are true:

(a) T(Mi)='union of' for each Mi in this class.

(b) There is an n such that $1 \leq n \leq L(Mi)$ for the Mi in the class and for all Mi, Mj in the class, for all m, $1 \leq m < n$, $S(Mi,m)=S(Mj,m)$.

(c) For the n of (b), there are Mi and Mj in the class such that $S(Mi,n) \neq S(Mj,n)$.

(d) For any Mi in the class, and any Mj in another class satisfying (a), (b) and (c), $C(Mi) < C(Mj)$.

If there is no class satisfying the above, terminate the algorithm. Otherwise partition the class as follows.

(a) For each Mi for which $C(Mi) > C(Mj)$ for any Mj in the considered class, redefine C(Mi) to be one greater than it was before this elaboration of step 5 was commenced.

(b) Find the value of S(Mi,n) which is minimal for the above n and every Mi in the considered class. Redefine C(Mi) to be one greater than it was before this elaboration of step 5 was commenced for each Mi such that S(Mi,n) is greater than the found minimum.

Return to step 2.

On termination of the algorithm, duplicate modes in constituent subclasses of unions can be eliminated. The modes Mi and Mj can now be ordered by saying Mi<Mj if and only if C(Mi)<C(Mj), and Mi=Mj if and only if C(Mi)=C(Mj).

2.3 Correctness of the Algorithm

Four properties of the preceeding algorithm will be demonstrated. First, the procedure terminates and is therefore indeed an algorithm. Second, the classes produced are equivalence classes. Third, the relative ordering imposed on any given pair of modes is independent of anything other than the modes themselves. And fourth, the ordering given is transitive, so that any given set of modes can be reduced to a unique linear ordering of nonequivalent modes. These are the required properties of the algorithm.

The operation of the procedure on classes of modes is that of partitioning those classes. For the algorithm to return to step 2, one of the steps 3, 4 or 5 must be performed successfully. Each of these steps properly divides a class, without moving any mode out of its previous order in the set of modes being ordered. Inasmuch as partitioning of a finite set is a finite process, the process will terminate whenever applied to a finite set of

modes.

Two modes are equivalent if the T function applied to each of them gives the same value, they have the same number of distinct constituent modes, the tags on their fields are correspondingly the same, and their distinct immediately constituent modes are correspondingly equivalent. The steps of the algorithm explicitly ensure that at termination no class contains modes which can be distinguished by these criteria.

Consider any given pair of modes which are not equivalent. Any decision by the algorithm to separate these two modes or any of their constituents is made strictly on the relative properties of the considered modes. Any modes not constituent to the considered modes do not affect the decision. Therefore each such decision is unique, and the operation that partitions the considered pair of modes will always divide them in the same order.

By extension of the argument of the last paragraph, the operations that partition any given subset are unique, and so any finite set of modes is uniquely ordered in a transitive manner.

CHAPTER III

Applications of the Algorithm

3.1 Representations of United Modes

There are four types of mode-dependent representations in an ALGOL 68 implementation: the declarers of an ALGOL 68 source program, the representation of modes in the compiler's tables, the representation of mode information on external media, and those data objects at run-time that require mode-dependent processing of a sort that cannot be uniquely determined at compile-time. These four types of representations require successively less processing by whatever part of an implementation receives each of them as input.

The basis for the representations used in this Chapter is the fact that each union can be represented by a uniquely ordered set of moods.

3.2 Compile-Time Handling of Modes

The algorithm of Chapter II is applied during or immediately after the pass of an ALGOL 68 compiler which

completes processing of the mode- declarations and declarers in the input source program. A compiler which allows mode-declarations after use as provided for in the full language must scan the source program, or some intermediate representations thereof, at least three times to be able to extract enough information to properly translate that program. All declarers cannot be discovered until two passes have been made over the source text. This can be shown by the example of a proper particular-program

```
begin a;
      a a;
      mode  a = int;
      skip
end .
```

It cannot be known that the "a" in the declaration "a a" is not a monadic-operator[R4.3.1e] until the second scan of the text. The algorithm produces information on mode equivalence which is required by the coercion processor[R8.2] and code generator in the compiler. Inasmuch as these operations can be performed no earlier than during the third scan of the source text, the ideal time to elaborate the algorithm is between the second and third scans.

A compiler which translates the source text in one pass needs to process some coercions before all the source program has been scanned. It is therefore inappropriate to

use an algorithm, such as that in Chapter II, which processes all the modes in a particular-program at one time. The principle applied in that algorithm can be used in a method of incrementally building a mode table, which is intrinsically faster than other methods proposed for processing modes in a one-pass compiler as will be shown in the next Chapter. The implementation of the algorithm and the application of its methods to a one-pass compiler is also discussed in Chapter IV.

Apart from the above-mentioned contribution of the algorithm to mode-list handling in a one-pass compiler, and the greater speed with which an ordered mode-list can be searched for a known mode, there seems to be no advantage to the compiler in using the algorithm of Chapter II rather than, say, that of Zosel. The order imposed on the modes has no relation to any sequence which may be the result of coercion, and therefore can give no aid to that process. The algorithm's application does, however, leave the mode-list in a state suitable for the processing of coercions. Almost all modes that can occur in coercion sequences are represented in the source program by declarers, denotations[R5.0.1a] or the use of operators or identifiers from the standard prelude[R10]. The exception is the mode transformation that can be produced by slicing[R8.6.1.1a] or selection[R8.5.2.1a]. Consider, for example, the particular-program

```
begin   ref[1:2]int  a = loc[1:2]int := (1,2);
        int  b = a[1];
        skip
end .
```

The unit "a[1]" is of mode ref int, a mode which does not occur explicitly in the program. However, this and all similar constructions have the properties that "extra modes", such as the ref int above, only occur in the intermediate stage of a coercion sequence, and that the extra modes are all of the form ref M, where M is a mode which is represented somewhere in the particular-program by a declarer. The problem has two possible solutions. The coercion process can check for equivalence of modes by checking if either they are both in the ordered mode-list and equivalent by the algorithm, or they are both 'reference to' some two modes which are in the mode-list and are equivalent. The second solution is to add to the set of modes under consideration all "extra modes" which could be produced from the modes in the table. This must be done before the algorithm is elaborated.

3.3 Intermediate Representation of Particular-Programs

The general computer user is concerned with three representations of an ALGOL 68 program: the source program, the intermediate representation of the program which is output by the compiler, usually referred to as the object

program, and the machine-code representation of a program that is loaded into the memory of the computer and is suitable for direct elaboration. The intermediate representation may be dispensed with if the ALGOL 68 compiler initiates elaboration directly. It may be suitable for loading and elaboration in a very simple and direct manner. On the other hand, the intermediate representation may require considerable processing before it is suitable for loading.

One case in which considerable processing of the intermediate representation is required is where different parts of the particular-program to be elaborated are compiled independently. Independent compilation is a technique which makes the modification of programs and use of subroutine libraries easier and more economical than if complete re-compilation were required each time a program was changed. Since its introduction in FORTRAN, the first widely used higher-level programming language, independent compilation has come to be supported by almost all computer installations. The chief device by which independent compilation has been supported is the symbolic resolution of references by one compiled module to the memory locations occupied by another. This allows transfer of control of elaboration between the modules at run-time. It is sufficient for linking FORTRAN routines but to previous authors seemed to be deficient for linking ALGOL 68

routines.

The problem stems from the fact that there is a countably infinite number of distinct modes which are available for representation in a proper ALGOL 68 particular-program. This fact does not especially complicate any aspect of code generation except the representation of objects whose mode could change during run-time elaboration, here referred to as "united objects". The proposed solution has been to represent each distinct mode appearing in a source program by a unique index, and appending the appropriate index to each united object at run-time to allow the identification of its current mode [3, 5]. The infinite number of possible modes, and the lack of direct communication between independently compiled program modules at compile-time prevents the unique determination of mode indices. This implementation of united objects requires, for the linking of independently compiled routines, that the program which does the linking resolve the representations of the mode indices which may occur in united objects accessible to more than one independently compiled program module [4, 6].

Apart from the construction of such a linking program, three solutions are available. Requiring that the compiler have access to all compiled modules, that can be referenced by the module that is being compiled, allows the compiler

itself to resolve representations of mode indices by compiling the new module to comform to the conventions of the older ones. This method has been used with success [11], but has certain disadvantages. It may be impractical for the computer user to supply the required modules. The method prevents the changing of modules used by a program without recompilation, thus losing one of the chief advantages of independent compilation. It imposes a fixed order of compilation of modules which prevents, for example, the use of an object defined in a main program by an independently compiled subroutine.

Another solution to the problem of representing mode indices is to use as the index a canonical representation of the mode involved. The total number of modes being countable, for each valid mode, there must be some particular-program which has as its first declarer a representation of that mode and which is the "smallest" such particular-program by some fixed ordering sequence. Such a sequence could be created by the comparison of the source representations of particular-programs as character strings. This canonical representation is generally much too large to be of any practical use as a mode index for run-time united objects. Such canonical representations are intrinsically large, as will be shown in section 3.5. However, that section will also suggest a practical method of deriving these canonical representations, and suggest a use for them.

The only representation of mode information that is needed at run-time is that used to distinguish the current mode of a united object. In an ALGOL 68 program this is the only situation where resolution of data representations between independently compiled modules would ever be required. The run-time representation of united objects described in section 3.4, takes recognition of the fact that for any united object, the compiler has available a description of those modes of which that object can be an instance of a value. The representation therefore need only distinguish between the modes of the object, and not between all modes in the particular-program under consideration. The algorithm of Chapter II allows the compiler to order uniquely the appropriate modes for a united object and choose a unique set of indices based on this ordering. Therefore no special processing of the intermediate representation of an ALGOL 68 program is required.

The above statement does not mean to imply that the facilities provided by current linking programs are satisfactory. ALGOL 68 is notable in that a compiler for the language is capable of checking for almost all erroneous and inconsistent uses of data objects, but the compiler cannot check to see that modes of corresponding data objects in independently compiled modules are equivalent. Furthermore, an optional feature of the compiler may produce

a module that can be reasonably linked only to modules which have been compiled with that feature enabled. Especially in the use of subroutine libraries, errors that would be caught by checks for such inconsistencies have a high probability of occurence. Language-dependent checks could be performed by allowing checking routines to be input to the linking program and by providing a method for the linking program to use these routines at the appropriate times.

Most current linking programs could be modified to include a general mechanism for the comparison of various strings which would be input for the purpose of ensuring consistency of program features. Such a mechanism could be used to check for consistent use of objects in independently compiled modules and for consistent use of optional scope checking. ALGOL 68 dependent processing of intermediate program representations is therefore probably not required.

3.4 Run-Time Handling of Modes

An ALGOL 68 compiler can, for any particular-program, determine what machine language code is required to perform all run-time data manipulation. The only case in which there is any uncertainty regarding which segment of code is to be elaborated is the copying of a united object or the checking of its mode. The copying of a united object must fall into one of three classes: copying objects whose modes

have been united to[R8.2.4] more inclusive ones, copying united objects without any change in the inclusiveness of the union, and copying united objects as a result of a conformity operation, and so reducing the inclusiveness of the union. In all types of such copying the method of implementation is the same: a routine appropriate to the mode of the object being copied must be chosen to perform the operation, and transfer of control must then be passed to that routine. In a conformity relation, transfer of control may pass to an exit if the relation fails.

Application of the algorithm of Chapter II produces a very fast and efficient method of selecting the appropriate routine. Assigning an index to each mode in a particular-program and using this index to identify the mode of united objects produces a significantly slower and less efficient implementation. Using the latter method requires that when selecting a routine each possibility for the mode of the object being copied has to be checked separately. This requires a table of mode indices which are checked against the index of the considered object. Use of the table requires the overhead of a program loop and the handling of a pointer to the table. A conformity relation which requires no copying still requires the use of a loop to check for the truth of the relation.

Identifying the mode of a united object by the index

of its mode in a list of its possible modes, allows the index to be used to access a table of routine addresses directly. Thus the correct routine is selected quickly and only a table of addresses is required. In a machine whose address unit is less than its word size, for instance in a character addressed machine, the mode indices can be kept as multiples of the word address increment. This further speeds the operation. The range of values of the index in each united object is known at compile time. Thus the storage it occupies can readily be chosen to optimize either the speed or storage requirements of the particular program. A useful special case of optimizing storage requirements is that of a union with two moods. In this case the mode index need occupy only one bit of storage. Testing this bit directly can produce a very fast method of selecting an appropriate course of action, which does not require an inconsistent method of representation.

It has been suggested [12] that the conformity relation[R8.3.2.1a] and the conformity case clause[R9.4] are not satisfactory mechanisms for examining and extracting the value of united objects. To check whether or not a tertiary[R8.1.1b] "T" is an instance of a value of mode "t", for example, the unit "loc t :: T" must be used. Involved is the totally wasted expense of the local generation "loc t". This case is easy to detect and optimize, but is confusing for the user, especially in the case that the mode

involved has some bounds in it which must be specified in spite of the fact they are unused. A more difficult example is

```
begin a a, b b, c c;
      case a, b, c ::= U in
           A, B, C out Z esac
end .
```

Where U is a tertiary of a mode which is a union of a, b, c and possibly some other modes, and A, B, C, and Z are units to be elaborated using the value of U if U is of mode a, b, c or none of these, respectively. Apart from being clumsy, the construction again requires the local generations used tc produce a, b and c, even though they may not be used as variables in A, B and C. Moreover one or two of the identifiers may not be used in the body of the case statement, making the copying operation in the conformity relation potentially superfluous.

A formulation of the conformity case clause has been suggested [13] which answers these criticisms and is syntactically easier for the compiler to handle. In the new formulation the above example would be written as

```
casec U in
      (a a): A,
      (b b): B,
      (c c): C out Z cesac .
```

The alternative which is selected is elaborated so that the

appropriate identifier, if any, is made to possess[R2.2.2g] a new instance[R2.2.1] of the value of the tertiary U, it being of the correct mode, of course. No extraneous local generators[R8.5.1.1b] are needed, and if, for example, the value of U is not required, to elaborate B, then the second alternative can be written as "(*b*): B", eliminating the need to copy the value of U in this case. The method of compiling the new conformity case is very straight-forward. The union U is elaborated, and a branch table formed to direct its alternate results.

Another example illustrates an interesting anomaly. Consider the two particular-programs

```
begin int  i;
      i := 2;
      case  i  in
            skip,  skip,  skip   esac
end and
begin union(int,real,bool) u;
      u := 2.0;
      casec  u  in
            (int):  skip ,
            (real):  skip ,
            (bool):  skip   cesac
end .
```

On most machines the case clause[R9.4c,d] in the second example will execute faster than in the first. This is

because the integer used as an index may need more manipulation before it is suitable as a table index, and in any case it has to be checked to see if it has a value other than 1, 2 or 3. In the second case, the index in the united object "u" can have only one of three values. This suggests that a method of specifying the range of values and use of a integer may have some application in a higher-level programming language.

The indexed method of implementing unions makes their handling a highly efficient operation, justifying their introduction into ALGOL 68 and assuring their place in future programming languages.

3.5 Transput of United Objects

Each valid ALGOL 68 mode consists of a "terminal" (the T(Mi) of Chapter II), and zero or more constituents, which are themselves modes. This construct corresponds to the "List" of Knuth [17, page 312]. The constituents of each mode have a unique ordering, provided by either their definition in the case of non-united modes, or by the algorithm of Chapter II in the case of united modes. Any List whose constituents are ordered can be given a unique linear representation similar to that described below for ALGOL 68 modes.

Label each node in the List representation of the mode

under consideration. This can be done uniquely as there is a starting node, that of the mode under consideration, and the offspring of each node are uniquely ordered. The mode can now be represented by a linear representation of the List with all but the first instance of each node represented by the label of that node.

As an example, consider

```
union form = (ref const, ref var, ref triple, ref call);
struct const = (real value);
struct var = (string name, real value);
struct triple = (form left operand, int operator,
    form right operand);
struct function = (ref var bound var, form body);
struct call = (ref function function name,
    form parameter); .
```

Application of the mode-ordering algorithm of Chapter II will order the constituents of the union form as ref const, ref var, ref call, ref triple. So after ordering and labeling of modes in a left-hand first order, the set of modes is

```
1: union(2, 5, 9, 13)                    {form}
2: ref 3
3: struct(4 value)                       {const}
4: real
5: ref 6
6: struct(7 name, 4 value)               {var}
```

```
 7: rowof 8
 8: char
 9: ref 10
10: struct(11 function name, 1 parameter)        {call}
10: ref 12
12: struct(9 bound var, 1 body)                  {function}
13: ref 14
14: struct(1 left operand, 15 operator, 1 right operand)
                                                 {triple}
15: int .
```

The above reduces to the canonical form for the mode form:

```
union(
  ref struct(real value),
  ref struct(
    rowof char name,
    4 value),
  ref struct(
    ref struct(
      5 bound var,
      1 body) function name,
    1 parameter),
  ref struct(
    1 left operand,
    int operator,
    1 right operand)) .
```

Note that the labels need not be included in this final

form, as they can be readily regenerated by counting through the modes in the canonical form.

The above example illustrates that a constituent of a canonical form is not necessarily the canonical form of a constituent. To extract mode information from a canonical form, therefore, the List representation of the mode must be regenerated.

It will be noticed immediately that the above form is rather large, and due to the presence of field-tags, which can be represented in a no more compact form, cannot be significantly reduced in size. Such a representation is therefore only of use where a canonical representation is required for modes. In any case the amount of manipulation that such a form undergoes should be reduced to a minimum. There are two uses of such a form, both of which require mode information to be stored on external media. Firstly, in the intermediate representation of an independently compiled module of a particular-program, each identifier which is accessible to some other module, or is accessed from some other module, can have associated with it its canonical form. This allows a linking program to check for compatibility of use of that identifier in all modules. Secondly, the mode of objects stored on backing media can be represented in canonical form, so increasing the flexibility of the transput[R5.5.1aa] facility in ALGOL 68 [14].

The representation of external media as arrays of mode int is very inappropriate to many applications. It has three main disadvantages: no data type other than int, real, bool, char or some multiple or structure[R2.2.3] thereof can be transput; multiple data types lose their structure when transput; and the representation of file structure, especially that on direct-access devices, is severely limited, reducing both the utility and efficiency of the use of backing media. One solution to these problems is to generalize a file to allow it to be of any mode. This allows both simple files more realistically, such as [,]char for an alphanumeric card reader, and complex files, such as a multi-level structure for a symbolically indexed direct access file. There is great advantage in the transput of non-primitive data types, as is discussed below.

The ability of an arrays to retain the bound information in its descriptor on backing media would reduce the housekeeping activities of computationally-oriented programs with otherwise straight-forward transput requirements.

The placing of routines on backing media would only be a recognition of the standard practice of keeping subroutine libraries. The only limitation on the transput of procedure objects is that they cannot have full generality of external references. This limitation has the general formulation

that backing media have a globality greater than that of any particular-program. Procedure objects are therefore limited in transput by the scope restrictions in ALGOL 68. In the same way the transput of formats can be reasonably defined, and the transput of references prevented.

The fact that procedure objects can be placed on external media doesn't eliminate the usefulness of linking programs. The types of external references allowed routines which can be transput are too limited to provide for all the facilities available with independent compilation. The class of routines which can be transput is, however, a useful one, being essentially those routines with no side effects at the level of globality of a particular-program.

The creation of files with united constituent modes not only allows the transput of unions but allows the creation of files with records of varying formats. On input the mode of the object on the backing medium could be readily discovered. Not only would the placing of a canonical form of the mode of a file allow checking for errors in its use, but it would also allow access by a user unaware of all the data types stored on a medium. By a mechanism similar to the conformity relation a program could access those records of whose data type it was aware, and be informed of attempts to access other data types. Such a facility would be useful for accessing a data base whose

form developed over a lcng period of time.

Some reasonable control over the structure of a data base is a requirement in many business and information retrieval applications. The array structure of files in ALGCL 68 makes the construction of one or more levels of symbolic indexing highly impractical. For instance, insertion of records into the middle of files requires a very inefficient simulation. A list-structured object would be much more appropriate. Reference data types are well suited to represent certain methods of record linkage in files. The particular-program must be prevented from accessing these references, as their representation would be quite incompatible with that of references within a particular-program. For reasons of efficiency and compatibility with external standards, the representation of many data types on external media could differ from that within a particular-program. Therefore installation-defined routines would be needed to buffer the transput operation. This practice could be facilitated by the separation of formatting from transput. After all, formatting is the process of transfcrming data objects to and from character strings, with no direct reference to transput.

The introduction of a practical method of representing united objects on external media allows the structure of files to be generalized, so greatly improving the ability of ALGCL 68 programs to communicate with the outside world.

CHAPTER IV

Implementation of the Algorithm

4.1 General Considerations

The algorithm described in Chapter II need be implemented only in the ALGOL 68 compiler. All other components of an ALGOL 68 implementation can use the mode information as processed by that algorithm. The savings involved are considerable, inasmuch as processing the output of the algorithm, either in the form of mode indices in united objects, or of canonical forms, is very much less complex than processing its tree-structured input.

There is a large class of formulations of the algorithm. The crux of the algorithm is that for any set of modes, the same partitionings of the set be performed in the same order at each invocation of the algorithm. The formulation in Chapter II is one of the simplest, but a practical implementation would require some improvements for the sake of speed.

Initial collection of the declarers from the source program to provide modes on which the algorithm can operate

is intrinsically a far slower process than the operation of the algorithm itself. The speed with which declarers are discovered is dependent on the parsing technique. Building the table of modes to be processed, or mode-list, will be considered below.

The number of declarers in a particular-program is typically far larger than the number of distinct modes in that program. Most declarers can be eliminated by simple checks for the more obvious mode equivalences. These checks should be applied before entering a newly found declarer in the mode-list. Primitive modes have no criteria for equivalence besides the equality of their written representation. Therefore the primitive modes can be kept in a separate list on input with no need for duplicates. Two declarers also represent equivalent modes if they are both the same indicant in the same range. Except for the resolution of indicants, step 1 of section 2.2 can be completed while inputing the declarers by forming the initial classes based on the terminals of the modes, the T(Mi) of section 2.2, at that time. This can readily be taken to the extent of actually doing the ordering based on the T(Mi) on input. The advantage of ordering is that the appropriate place for a newly encountered mode in the mode-list can be much more readily found if a binary search of the list can be made. It is also practical on input to find equivalence between a newly entered mode and one in the list

by comparing the immediate constituents for known equivalence. The above simple checks will probably leave the algorithm very little or no work to do for most particular-programs. They take advantage of a programmer's inclination to write declarers in a consistent manner and to use indicants to represent complex modes. They also reduce the number of modes in the mode-list to very nearly the number of distinct modes in the program.

The above input method does most of the work associated with mode ordering for programs with a small number of simple modes. Programs with a significant number of complicated modes require a lot of work that cannot be done on input. It is just these programs, though, that benefit from the algorithm during the coercion phase of the compiler, and at run-time. Therefore an inefficient form of the algorithm, used for considerations of space or speed of implementation, detracts only slightly from its benefits, and only in proportion to its benefits.

During the application of the algorithm, the least recently examined class is the most likely to have criteria for partitioning. Continuing to scan for a class to partition after one has been found and partitioned, instead of restarting at the begining of the mode-list is therefore to be recommended. This change in the order of finding partitionable classes preserves the fixed order of

partitioning required by the algorithm. Likewise the sub-ordering of the immediate constituents of united modes can be postponed until those modes are found in a class being considered for partition. This saves the redundant scans of the mode-list described in step 2 of Section 2.2.

If the scanning of the mode-list is to be continued after a partitioning, then further care must be taken that the order in which partitioning operations are performed on any set of modes is independent of all other modes being processed at the same time. It is therefore required to partition a class using all possible criteria before progressing tc another class. A class should be partitioned into as many classes as required to ensure that all resultant classes contain only indistinguishable modes. This certainly ensures that there is no skipping over partitionings, caused by modes extraneous to any subset of those being processed. It also drastically reduces the total number of scans of the mode-list required to complete the algorithm.

No class of primitive modes can be partitioned by the algorithm. Any other class may be subject to a sequence of partitionings during the process of the algorithm. Any class which is the immediate result of partitioning, and contains any mode whose immediate constituents are members of classes which can definitely not be partitioned any

further, can itself definitely not be partitioned any further. A class which contains any such mode has that property for all its modes, so the test is easily made.

The form in which the mode-list should be kept during the application of the algorithm is as an ordered list of unordered sets of modes. Thus at any stage of the algorithm, any two modes are either certainly not equivalent or there has been no reason to suspect their non-equivalence. No optimizations can be made on the basis of suspicions of the relation between two modes.

The detection of all context-condition violations except those which would prevent the elaboration of the mode ordering algorithm should be postponed until after its elaboration. This prevents the duplication of checks on equivalent modes. The only form of mode declaration which would adversely affect the algorithm would be those in which an indicant is defined as being itself, such as

mode a = b; mode b = a; .

These cases can be readily detected in the first step of the algorithm. The output of the algorithm is a mode-list of the same sort as Zosel's, except that it is ordered. All context conditions can therefore be readily checked for.

4.2 Special Considerations for One-Pass Compilers

Compilers which must generate object code for part of

a particular-program before reading all of the source text, must usually perform coercion operations on some coercends before all the declarers in the program have been encountered. A one-pass compiler can handle only a subset of ALGOL 68 which requires definition of all indicants before use of declarers containing them as constituents. The exact limitations on such a subset of the language have been discussed elsewhere [15].

The requirements of a one-pass compiler make inappropriate an algorithm which equivalences or orders all the modes in a particular-program at once. The equivalencing algorithm of C.H.A. Koster [16] can be used, but its formulation makes it undesirable in one-pass compilers, where speed and space usage are usually overriding considerations in the compiler's design. The method of the mode ordering algorithm yields a technique for incrementally constructing a mode table and is also quite fast.

The technique is essentially to reapply the algorithm to all known modes whenever mode equivalence information is needed. This apparently wasteful method can be optimized to the extent of making it probably the fastest mode-list handling technique proposed. All the optimizations of section 4.1 can be applied to a one-pass compiler. The algorithm need not be used when no new modes have been added

to the list since its last application. The fact that the modes already subject to the algorithm are ordered and non-redundant can be used to prevent regeneration of information already known.

The information that the modes already in the mode-list, or old modes, are ordered can be used in two ways. All modes which fall into classes requiring no new entries to the mode-list, or contituents thereof, need no further processing, as their mutual relative ordering and their ordering with respect to all modes outside the class is determined by a prvious invocation of the algorithm. The ordering relation between any new mode in a class and any old mode in that class can be used to determine immediately the new mode's relation to some or all of the other old modes in that class. Thus it can quickly be determined where to partition a class.

A class which contains only old modes, two or more of which are constituents of new modes, must go through the steps that would be required if those constituent modes had been new modes. This is to retain the vital order of partitioning which is the crux of the algorithm. Old modes and new modes in the same class must be partitioned in the regular manner in order that their proper relation can be determined.

By taking advantage of the transitivity of the mode-

ordering relation, the insertion of new modes into an ordered mode list can be made faster. One-pass compilers are traditionally designed for compile-and-go applications, but the ordering of modes also makes it practical to include the facility of independent compilation in their features.

CHAPTER V

Conclusion

An algorithm has been presented which can be implemented in an ALGOL 68 compiler as an inexpensive procedure for uniquely ordering its internal table of modes. Methods have been described by which the ordering of modes can be made to speed the operation of both the compiler and the object programs it produces. Ordering modes also makes implementation of existing features of ALGOL 68 easier, and implementation of new features practical.

There are four ways in which ordered modes aid the implementation of ALGOL 68. Firstly, the handling of mode tables in the compiler is made more efficient, enabling the compiler to perform its searches of them faster. Secondly, the handling of united objects at run-time can be made more efficient by using the order of the moods of the object's modes to address tables directly. Thirdly, the uniqueness of the ordering provides a unique representation of united objects, thus eliminating the only property of the language's implementation which otherwise requires special processing by linking programs. Fourthly, a unique and

efficient representation of mode information is made available by ordering, which makes possible a much more flexible and powerful formulation of the transput facility. The value of such a facility would be further enhanced by the establishment of a standard mode ordering which would allow the construction of data files with formats independent of particular ALGOL 68 implementations.

Higher-level languages have been criticized for their inefficient handling of data structures, and for the difficulty of incorporating them into the facilities of a computer installation. The application of mode ordering contributes significantly to the invalidation of both arguments, and should therefore further the utility and acceptance of higher-level programming languages.

REFERENCES

1. Van Wijngaarden, A. (editor), Mailloux, B.J., Peck, J.E.L., and Koster, C.H.A., Report on the Algorithmic Language ALGOL 68, Report MR101, Mathematisch Centrum, Amsterdam, 1969.

2. Naur, P. (editor), "Revised Report on the Algorithmic Language ALGOL 60", Communications of the ACM, VI, (January, 1963), p. 1.

3. Mailloux, B.J., "On the Implementation of ALGOL 68", (Ph.D. Dissertation, Mathematisch Centrum, Amsterdam, 1968).

4. Goos, G., "Some Problems in Compiling ALGOL 68", in ALGOL 68 Implementation, (North Holland, 1971), p. 179.

5. Wilmott, S.J., General Storage Organization of an ALGOL 68 Program, IFIP Working Group 2.1 Paper (Manchester 12) 152.

6. Shapiro, V., "Combining Independently-Compiled ALGOL 68 Programs", (M.Sc. Thesis, University of Alberta, 1972).

7. Lindsey, C.H., and van der Meulen, S.G., Informal Introduction to ALGOL 68, (North Holland, 1971).

8. Peck, J.E.L., An ALGOL 68 Companion, (Department of Computer Science, University of British Columbia, 1972).

9. Zosel, M.E., "A Formal Grammar for the Representation of Modes and its Application to ALGOL 68",(Ph.D. Dissertation, University of Washington, 1971).

10. Peck, J.E.L., "On Storage of Modes and Some Context Conditions", in Proceedings of an Informal Conference on ALGOL 68 Implementation, (Department of Computer Science, University of British Columbia, 1969), p. 70.

11. Currie, I.F., Bond, S.G., and Morison, J.D., "ALGOL 68-R", in ALGOL 68 Implementation, (North Holland, 1971), p. 21.

12. The Subcommittee on Maintenance and Improvement, Report on Considered Improvements, IFIP Working Group 2.1 Paper (Fontainbleau 9) 192.

13. Thomas, L.K., and Wilmott S.J., Comments on Current Proposals being Considered by the Subcommittee on Maintenance and Improvement (Revised), IFIP Working Group 2.1 Paper (Fontainbleau 3) 186, (June, 1972).

14. Lindsey, C.H., Koster, C.H.A., Jenkins, D.P., and Gilinsky, R.J., Report of the sub-committee on Data-processing and Transput, IFIP Working Group 2.1 Paper (Manchester 11) 151.

15. Thomas, L.K., "ALGOL 68 - Considerations for a One-Pass Implementation", (M.Sc. Thesis, University of Alberta, 1972).

16. Koster, C.H.A., "On Infinite Modes", ALGOL Bulletin, No. 30, (February, 1969), p. 86.

17. Knuth, D.E., The Art of Computer Programming, (Addison-Wesley, 1968) Volume 1 (Fundamental Algorithms).

APPENDIX A

```
BEGIN COMMENT

     The following program is an implementation of the
algorithm of Chapter II. It inputs modes in a form similar
to that of an ALGOL 68 mode-declaration. On command it
outputs the canonical form of the mode of any indicant
that has been defined or outputs the defined indicants in
the order determined by the algorithm.

     The program is written in ALGOL-W, which is
described in ALGOL-W Reference Manual by R.L. Sites,(Computer
Science Department, Stanford University, 1971). Examples
of the program's use follow the listing.

     The records defined below represent the compiler's
internal tables. MODE holds an entry in the mode-list,
representing one mode. Its fields are

MODEKY          terminal symbol of mode (e.g. int, ref,
                procvoid)
MODELN          L(Mi) of Chapter II
MODENF          N(Mi) of Chapter II
MODECLS         C(Mi) of Chapter II
MODEPR          pointer to the fields or constituents of Mi
MODEBK,MODEFW   linkage for the mode-list.

PRAM holds an entry for a field of a mode described in a
MODE record. Its fields are

PRAMMD          constituent mode of field
PRAMTG          field-tag if this is a field of a structure
PRAMLK          linkage for the fields of a mode.

TAG holds part of a variable length string used as an
indicant or field-tag. TABLE holds an entry for an indicant.
INDICANTS points to the table of indicants and MODES points
to the mode-list.  ;

    RECORD MODE (STRING(6) MODEKY;
                 INTEGER MODELN,MODEPLC,MODENF,MODECLS;
                 LOGICAL MODEMK;
                 REFERENCE(PRAM,TAG) MODEPR;
```

```
            REFERENCE(MODE) MODEBK,MODEFW);
    RECORD PRAM(REFERENCE(MODE) PRAMMD;
            REFERENCE(TAG) PRAMTG;
            REFERENCE(PRAM) PRAMLK);
    RECORD TAG(STRING(16) TAGNM;
            REFERENCE(TAG) TAGLK);
    RECORD TABLE(REFERENCE(TAG) TABLENM;
            LOGICAL TABLEMK;
            REFERENCE(MODE) TABLEMD;
            REFERENCE(TABLE) TABLELK);
    REFERENCE(TABLE) INDICANTS;
    REFERENCE(MODE) MODES;
    INTEGER NUMBINDIC;

    LOGICAL ORDERFLAG;

COMMENT
     ORDER is the implementation of the algorithm of
Chapter II.  ;

    PROCEDURE ORDER;
    IF(¬ORDERFLAG)AND(MODES¬=NULL)THEN
    BEGIN REFERENCE(MODE) M,MM,MT,MTT,MBK,MFW;
          INTEGER N;

COMMENT Step 1:
     Replace the indicants with their definitions, unravel
the unions, and form classes on the basis of the T(Mi) of
Chapter II.  Note that the modes in the mode-list are
partially ordered by GTPRIO on input (in the routine
ENTER).  ;

          UNTABLE;
          UNRAVEL;
          M:=MODES;
          N:=1;
          MM:=NULL;
          WHILE M¬=NULL DO
          BEGIN MODECLS(M):=N;
                MODEBK(M):=MM;
                MM:=M;
                MODEMK(M):=FALSE;
                IF MODEFW(M)¬=NULL THEN
                IF GTPRIO(MODEFW(M),M) THEN
                N:=N+1;
                M:=MODEFW(M)
          END;

COMMENT Steps 2 to 5:
     Order the moods of unions, then search for a class
that can be partitioned on the criteria of steps 3, 4, and
5 of the algorithm.  ;
```

```
WHILE(BEGIN ORDERMOODS;
      M:=MODES;
      WHILE(BEGIN MBK:=MODEBK(M);
            MFW:=MODEFW(M);
            MODEFW(M):=NULL;
            MM:=NULL;
            WHILE IF MFW=NULL THEN FALSE ELSE
                  MODECLS(MFW)=MODECLS(M) DO
            BEGIN MT:=MFW;
                  MFW:=MODEFW(MFW);
                  MODEFW(MT):=MM;
                  MM:=MT
            END;
            MT:=NULL;
            WHILE MM¬=NULL DO
            IF GTPOST(M,MM) THEN
            BEGIN MT:=MODECAT(MT,M);
                  M:=MM;
                  MM:=MODEFW(MM);
                  MODEFW(M):=NULL
            END ELSE
            IF GTPOST(MM,M) THEN
            BEGIN MTT:=MODEFW(MM);
                  MODEFW(MM):=MT;
                  MT:=MM;
                  MM:=MTT
            END ELSE
            BEGIN MTT:=MODEFW(MM);
                  MODEFW(MM):=M;
                  M:=MM;
                  MM:=MTT
            END;
            IF MBK=NULL THEN MODES:=M ELSE
            MODEFW(MBK):=M;
            WHILE (BEGIN MODEBK(M):=MBK;
                  MBK:=M;
                  M:=MODEFW(M);
                  M¬=NULL
            END)DO;
            MM:=MT;
            MODEFW(MBK):=MT;
            WHILE MM¬=NULL DO
            BEGIN MODEBK(MM):=MBK;
                  MBK:=MM;
                  MM:=MODEFW(MM)
            END;
            MODEFW(MBK):=MFW;
            IF MFW¬=NULL THEN
            MODEBK(MFW):=MBK;
            M:=MFW;
            (M¬=NULL)AND(MT=NULL)
```

```
                END)DO;
                MTT:=MT;
                IF MT¬=NULL THEN
                BEGIN WHILE MT¬=NULL DO
                      BEGIN MODECLS(MT):=MODECLS(MT)+1;
                            MT:=MODEFW(MT)
                      END
                END;
                MTT¬=NULL
          END)DO;
          CRUNCH;
          ORDERFLAG:=TRUE
    END;

COMMENT
      CRUNCH deletes all duplicate modes, deletes duplicate
fields from united modes, and makes indicants point to the
representative of their defined mode.  ;

    PROCEDURE CRUNCH;
    BEGIN REFERENCE(TABLE) I;
          REFERENCE(MODE) M,MM;
          REFERENCE(PRAM) P;
          INTEGER N;
          I:=INDICANTS;
          WHILE I¬=NULL DO
          IF MODEKY(TABLEMD(I))¬="TAG   " THEN
          BEGIN M:=TABLEMD(I);
                N:=MODECLS(M);
                WHILE IF MODEFW(M)=NULL THEN FALSE ELSE
                      MODECLS(MODEFW(M))=N DO
                M:=MODEFW(M);
                TABLEMD(I):=M;
                I:=TABLELK(I)
          END ELSE
          I:=TABLELK(I);
          M:=MODES;
          WHILE M¬=NULL DO
    BEGIN P:=MODEPR(M);
                WHILE P¬=NULL DO
                IF MODEKY(PRAMMD(P))="TAG   " THEN
                P:=PRAMLK(P) ELSE
                BEGIN MM:=PRAMMD(P);
                      N:=MODECLS(MM);
                    WHILE IF MODEFW(MM)=NULL THEN FALSE ELSE
                          MODECLS(MODEFW(MM))=N DO
                      MM:=MODEFW(MM);
                      PRAMMD(P):=MM;
                      P:=PRAMLK(P)
                END;
                IF MODEKY(M)="UNION " THEN
                BEGIN P:=MODEPR(M);
```

```
                    WHILE P¬=NULL DO
                    BEGIN WHILE IF PRAMLK(P)=NULL THEN
                            FALSE ELSE MODECLS(PRAMMD(P))=
                            MODECLS(PRAMMD(PRAMLK(P))) DO
                          PRAMLK(P):=PRAMLK(PRAMLK(P));
                          P:=PRAMLK(P)
                    END
              END;
              M:=MODEFW(M)
        END;
        WHILE IF MODEFW(MODES)=NULL THEN FALSE ELSE
              MODECLS(MODES)=MODECLS(MODEFW(MODES)) DO
        MODES:=MODEFW(MODES);
        M:=MODES;
        WHILE(BEGIN MM:=MODEFW(M);
                    MM¬=NULL
              END)DO
        BEGIN WHILE IF MODEFW(MM)=NULL THEN FALSE ELSE
                MODECLS(MM)=MODECLS(MODEFW(MM)) DO
              MM:=MODEFW(MM);
              MODEFW(M):=MM;
              M:=MM
        END
    END;

COMMENT
      GTTAG, like EQTAG below, compares two tags. GTTAG
returns true if the first tag is "greater than"
the second, and EQTAG returns true if they are equal. ;

    LOGICAL PROCEDURE GTTAG(REFERENCE(TAG) VALUE A,B);
    (BEGIN WHILE IF(A=NULL)OR(B=NULL)THEN FALSE ELSE
              TAGNM(A)=TAGNM(B) DO
          BEGIN A:=TAGLK(A);
              B:=TAGLK(B)
          END;
          IF A=NULL THEN FALSE ELSE
          IF B=NULL THEN TRUE ELSE
          TAGNM(A)>TAGNM(B)
    END);

    LOGICAL PROCEDURE EQTAG(REFERENCE(TAG) VALUE A,B);
    BEGIN WHILE IF (A=NULL)OR(B=NULL) THEN FALSE ELSE
              TAGNM(A)=TAGNM(B) DO
          BEGIN A:=TAGLK(A);
              B:=TAGLK(B)
          END;
          (A=NULL)AND(B=NULL)
    END;

COMMENT
      GTPRIO compares two modes on the criteria of step
```

1 of section 2.2. ;

```
    LOGICAL PROCEDURE GTPRIO(REFERENCE(MODE) VALUE A,B);
    IF MODEKY(A)="TAG   " THEN FALSE ELSE
    IF MODEKY(B)="TAG   " THEN TRUE ELSE
    IF MODEKY(A)>MODEKY(B) THEN TRUE ELSE
    IF MODEKY(A)<MODEKY(B) THEN FALSE ELSE
    IF MODEKY(A)="UNION " THEN FALSE ELSE
    IF MODELN(A)¬=MODELN(B) THEN MODELN(A)>MODELN(B) ELSE
    IF MODENF(A)¬=MODENF(B) THEN MODENF(A)>MODENF(B) ELSE
    IF MODEKY(A)¬="STRUCT" THEN FALSE ELSE
    (BEGIN REFERENCE(PRAM) P,Q;
          P:=MODEPR(A);
          Q:=MODEPR(B);
          WHILE IF P=NULL THEN FALSE ELSE
                EQTAG(PRAMTG(P),PRAMTG(Q)) DO
          BEGIN P:=PRAMLK(P);
                Q:=PRAMLK(Q)
          END;
          IF P=NULL THEN FALSE ELSE
          GTTAG(PRAMTG(P),PRAMTG(Q))
    END);
```

COMMENT
GTPOST, like GTPRIO, compares two modes, but based on the criteria used in steps 3, 4 and 5 to determine if two modes should be placed in different classes. ;

```
    LOGICAL PROCEDURE GTPOST(REFERENCE(MODE) VALUE A,B);
    IF MODEKY(A)="TAG   " THEN FALSE ELSE
    IF MODEKY(B)="TAG   " THEN TRUE ELSE
    IF MODECLS(A)¬=MODECLS(B) THEN MODECLS(A)>MODECLS(B)ELSE
    IF MODEKY(A)¬="UNION " THEN
    (BEGIN REFERENCE(PRAM) P,Q;
          P:=MODEPR(A);
          Q:=MODEPR(B);
          WHILE IF P=NULL THEN FALSE ELSE
                MODECLS(PRAMMD(P))=MODECLS(PRAMMD(Q)) DO
          BEGIN P:=PRAMLK(P);
                Q:=PRAMLK(Q)
          END;
          IF P=NULL THEN FALSE ELSE
          MODECLS(PRAMMD(P))>MODECLS(PRAMMD(Q))
    END)ELSE
    (BEGIN REFERENCE(PRAM) P,Q;
          INTEGER N;
          P:=MODEPR(A);
          Q:=MODEPR(B);
          WHILE IF(P=NULL)OR(Q=NULL)THEN FALSE ELSE
                MODECLS(PRAMMD(P))=MODECLS(PRAMMD(Q)) DO
          BEGIN N:=MODECLS(PRAMMD(P));
                WHILE(BEGIN P:=PRAMLK(P);
```

```
                    IF P=NULL THEN FALSE ELSE
                    MODECLS(PRAMMD(P))=N
              END)DO;
              WHILE(BEGIN Q:=PRAMLK(Q);
                    IF Q=NULL THEN FALSE ELSE
                    MODECLS(PRAMMD(Q))=N
              END)DO
        END;
        IF P=NULL THEN FALSE ELSE
        IF Q=NULL THEN TRUE ELSE
        MODECLS(PRAMMD(P))>MODECLS(PRAMMD(Q))
    END);

COMMENT
     ORDERMOODS orders the constituents of all united
modes in the set under consideration, and so implements
step 2 of section 2.2.  ;

    PROCEDURE ORDERMOODS;
    BEGIN REFERENCE(MODE) M;
        REFERENCE(PRAM) P,Q,R,S;
        M:=MODES;
        WHILE IF M=NULL THEN FALSE ELSE
              MODEKY(M)¬="UNION " DO
        M:=MODEFW(M);
        WHILE IF M=NULL THEN FALSE ELSE
              MODEKY(M)="UNION " DO
        IF MODEPR(M)¬=NULL THEN
        BEGIN P:=PRAMLK(MCDEPR(M));
              Q:=MODEPR(M);
              PRAMLK(Q):=NULL;
              WHILE P¬=NULL DO
              IF MCDECLS(PRAMMD(P))>MODECLS(PRAMMD(Q))THEN
              BEGIN R:=Q;
                    WHILE IF PRAMLK(R)=NULL THEN FALSE
                          ELSE MODECLS(PRAMMD(P))>
                          MODECLS(PRAMMD(PRAMLK(R))) DO
                    R:=PRAMLK(R);
                    S:=PRAMLK(R);
                    PRAMLK(R):=P;
                    P:=PRAMLK(P);
                    PRAMLK(PRAMLK(R)):=S
              END ELSE
              BEGIN R:=Q;
                    Q:=P;
                    P:=PRAMLK(P);
                    PRAMLK(Q):=R
              END;
              MODEPR(M):=Q;
              M:=MCDEFW(M)
        END ELSE
        M:=MODEFW(M)
```

```
    END;

    REFERENCE(MODE) PROCEDURE MODECAT
          (REFERENCE(MODE) VALUE A,B);
    IF A=NULL THEN B ELSE
    (BEGIN REFERENCE(MODE) M;
          M:=A;
      WHILE MODEFW(A)¬=NULL DO
          A:=MODEFW(A);
          MODEFW(A):=B;
          M
    END);

COMMENT
      UNTABLE replaces all uses of indicants in mode
definitions by their defined values, that is their modes.
At this point the unshielded mode definitions which would
adversely affect the algorithm are detected.  ;

    PROCEDURE UNTABLE;
    BEGIN REFERENCE(TABLE) I;
          REFERENCE(MODE) M,MM;
          REFERENCE(PRAM) P;
          INTEGER N;
          I:=INDICANTS;
          WHILE I¬=NULL DO
          BEGIN M:=TABLEMD(I);
                N:=0;
                MM:=NULL;
                WHILE(MODEKY(M)="TAG    ") AND
                      (N<NUMBINDIC) AND
                      (M¬=MM) DO
                BEGIN MM:=M;
                      N:=N+1;
                      M:=LOOKUP(MODEPR(M))
                END;
                IF N>=NUMBINDIC THEN
                BEGIN WRITE("***ERROR***  MODE ");
                      PRINTTAG(TABLENM(I));
                      WRITEON("=");
                      PRINTTAG(TABLENM(I));
                      WRITEON(";");
                      IOCCNTROL(2)
                END ELSE
                TABLEMD(I):=M;
                I:=TABLELK(I)
          END ;
          M:=MODES;
          WHILE M¬=NULL DO
          BEGIN P:=MODEPR(M);
                WHILE P¬=NULL DO
                BEGIN
```

```
            IF MODEKY(PRAMMD(P))="TAG   " THEN
            PRAMMD(P):=LOOKUP(MODEPR(PRAMMD(P)));
                  P:=PRAMLK(P)
            END;
            M:=MODEFW(M)
      END
END;

REFERENCE(MODE) PROCEDURE LOOKUP
      (REFERENCE(TAG) VALUE T);
(BEGIN REFERENCE(TABLE) SYMPT;
      SYMPT:=INDICANTS;
      WHILE IF SYMPT=NULL THEN FALSE ELSE
            ¬EQTAG(TABLENM(SYMPT),T) DO
      SYMPT:=TABLELK(SYMPT);
      IF SYMPT¬=NULL THEN TABLEMD(SYMPT) ELSE
     (BEGIN WRITE("***ERROR***  UNDEFINED INDICANT:");
            PRINTTAG(T);
            IOCONTROL(2);
            ENTER("TAG",0,T)
      END)
END);

COMMENT
      UNRAVEL and UNBOX unravel all united modes so that
all their constituents are non-united modes. ;

PROCEDURE UNRAVEL;
BEGIN REFERENCE(MODE) M,MM;
      M:=MODES;
      WHILE IF M=NULL THEN FALSE ELSE
            MODEKY(M)¬="UNION " DO
      M:=MODEFW(M);
      MM:=M;
      WHILE IF MM=NULL THEN FALSE ELSE
            MODEKY(MM)="UNION " DO
      BEGIN MODECLS(MM):=0;
            MM:=MODEFW(MM)
      END;
      WHILE IF M=NULL THEN FALSE ELSE
            MODEKY(M)="UNION " DO
      BEGIN MODEPR(M):=UNBOX(M);
            M:=MODEFW(M)
      END
END;

REFERENCE(PRAM) PROCEDURE UNBOX
            (REFERENCE(MODE) VALUE M);
IF MODECLS(M)=2 THEN MODEPR(M) ELSE
IF MODECLS(M)=1 THEN
(BEGIN WRITE("***ERROR***  MODE *=UNION(...,*,...);");
      IOCONTROL(2);
```

```
        MODECLS(M):=2;
        PRAM(M,NULL,NULL)
    END) ELSE
    (BEGIN REFERENCE(PRAM) P,Q;
        MODECLS(M):=1;
        P:=MODEPR(M);
        WHILE P¬=NULL DO
        BEGIN IF MODEKY(PRAMMD(P))="UNION " THEN
              BEGIN Q:=UNBOX(PRAMMD(P));
                    PRAMMD(P):=PRAMMD(Q);
                    WHILE PRAMLK(Q)¬=NULL DO
                    BEGIN Q:=PRAMLK(Q);
                          PRAMLK(P):=PRAM(PRAMMD(Q),
                                NULL,PRAMLK(P));
                          P:=PRAMLK(P)
                    END
              END;
              P:=PRAMLK(P)
        END;
        MODECLS(M):=2;
        MODEPR(M)
    END);

    INTEGER PLACE;

COMMENT
      PRINTMODE prints the canonical form of a mode using an
ordered mode-list to produce that form.  ;

    PROCEDURE PRINTMODE(REFERENCE(MODE) VALUE M);
    IF MODEKY(M)="INT    " THEN
     BEGIN PRINTSHONGS(MODELN(M));
           WRITEON("INT ")
    END ELSE
    IF MODEKY(M)="REAL   " THEN
     BEGIN PRINTSHONGS(MODELN(M));
           WRITEON("REAL ")
    END ELSE
    IF MODEKY(M)="BOOL   " THEN WRITEON("BOOL ") ELSE
    IF MODEKY(M)="CHAR   " THEN WRITEON("CHAR ") ELSE
    IF MODEKY(M)="FORMAT " THEN WRITEON("FORMAT ") ELSE
    IF MODEKY(M)="TAG    " THEN WRITEON("*UNDEFINED* ") ELSE
    IF(BEGIN PLACE:=PLACE+1;
             MODEPLC(M)¬=0
       END)THEN PRINTNUM(MODEPLC(M)) ELSE
    IF(BEGIN MODEPLC(M):=PLACE;
             MODEKY(M)="REF    "
       END)THEN
    BEGIN WRITEON("REF ");
          PRINTMODE(PRAMMD(MODEPR(M)))
    END ELSE
    IF(MODEKY(M)="ROW    ")OR(MODEKY(M)="ROWOF ") THEN
```

```
BEGIN WRITEON("(");
      WHILE MODEKY(M)="ROW    " DO
      BEGIN WRITEON(",");
            M:=PRAMMD(MODEPR(M))
      END;
      WRITEON(")");
      PRINTMODE(PRAMMD(MODEPR(M)))
END ELSE
IF MODEKY(M)="PROC   " THEN
BEGIN WRITEON("PROC ");
      PRINTPRAM(PRAMLK(MODEPR(M)));
      PRINTMODE(PRAMMD(MODEPR(M)))
END ELSE
IF MODEKY(M)="PRVOID" THEN
BEGIN WRITEON("PROC ");
      PRINTPRAM(MODEPR(M));
      WRITEON("VOID ")
END ELSE
IF MODEKY(M)="UNION " THEN
BEGIN WRITEON("UNION ");
      PRINTPRAM(MODEPR(M))
END ELSE
BEGIN WRITEON("STRUCT ");
      PRINTPRAM(MODEPR(M))
END;

PROCEDURE PRINTPRAM(REFERENCE(PRAM) VALUE P);
IF P¬=NULL THEN
BEGIN WRITEON("(");
      WHILE(BEGIN PRINTMODE(PRAMMD(P));
                  PRINTTAG(PRAMTG(P));
                  P:=PRAMLK(P);
                  P¬=NULL
            END)DO
      WRITEON(",");
      WRITEON(")")
END;

PROCEDURE PRINTNUM(INTEGER VALUE N);
BEGIN STRING(13) S;
      S(0|12):=INTBASE10(N);
      S(12|1):=" ";
      IF N<0 THEN
      BEGIN WRITEON("-");
            N:=-N
      END;
      IF N<10 THEN WRITEON(S(11|2)) ELSE
      IF N<100 THEN WRITEON(S(10|3)) ELSE
      IF N<1000 THEN WRITE(S(9|4)) ELSE
      WRITEON(S(2|11))
END;
```

```
PROCEDURE PRINTTAG(REFERENCE(TAG) VALUE T);
IF T¬=NULL THEN
BEGIN WHILE TAGLK(T)¬=NULL DO
      BEGIN WRITEON(TAGNM(T));
            T:=TAGLK(T)
      END;
      PRINTCHOP(TAGNM(T))
END;

PROCEDURE PRINTCHOP(STRING(16) VALUE S);
IF S(8|8)="        " THEN
BEGIN IF S(4|4)="    " THEN
      BEGIN IF S(2|2)="  " THEN
            BEGIN IF S(1|1)¬=" " THEN WRITEON(S(0|2))
             ELSE IF S(0|1)¬=" " THEN WRITEON(S(0|1))
            END ELSE
            IF S(3|1)=" " THEN WRITEON(S(0|3)) ELSE
            WRITEON(S(0|4))
      END ELSE
      IF S(6|2)="  " THEN
      BEGIN IF S(5|1)=" " THEN WRITEON(S(0|5)) ELSE
            WRITEON(S(0|6))
      END ELSE
      IF S(7|1)=" " THEN WRITEON(S(0|7)) ELSE
      WRITEON(S(0|8))
END ELSE
IF S(12|4)="    " THEN
BEGIN IF S(10|2)="  " THEN
      BEGIN IF S(9|1)=" " THEN WRITEON(S(0|9)) ELSE
            WRITEON(S(0|10))
      END ELSE
      IF S(11|1)=" " THEN WRITEON(S(0|11)) ELSE
      WRITEON(S(0|12))
END ELSE
IF S(14|2)="  " THEN
BEGIN IF S(13|1)=" " THEN WRITEON(S(0|13)) ELSE
      WRITEON(S(0|14))
END ELSE
IF S(15|1)=" " THEN WRITEON(S(0|15)) ELSE
WRITEON(S);

PROCEDURE PRINTSHONGS(INTEGER VALUE N);
IF N>0 THEN
BEGIN FOR I:=1 UNTIL N DO
      WRITEON("LONG ")
END ELSE
IF N<0 THEN
BEGIN FOR I:=1 UNTIL -N DO
      WRITEON("SHORT ")
END;

LOGICAL ERRORFLAG;
```

```
COMMENT
      INMODE reads in a mode and enters it and its
constituents in the mode-list. It uses none of the
optimizations of Chapter IV to aid the algorithm proper.
The routines for input are structured on three levels:
inputting a mode (by INMODE), inputting a symbol, tag,
or indicant (by INTAG), and inputting a symbol (by
LOOKCHAR). A simple BNF grammar for modes is

<mode>::= <shongsety> INT | <shongsety> REAL | BOOL |
          CHAR | FORMAT | <indicant> | REF <mode> |
          PROC <moid> | PROC(<pram>)<moid> | UNION(<pram>) |
          STRUCT(<fields>) | (<rowsety>) <mode>
<shongsety>::= <longs> | <shorts> | <empty>
<longs>::= LONG | LONG <longs>
<shorts>::= SHORT | SHORT <shorts>
<rowsety>::= <empty> | ,<rowsety>
<moid>::= <mode> | VOID
<pram>::= <mode> | <pram>,<mode>
<fields>::= <mode> <tag> | <fields>,<mode> <tag>
<tag>::= <alphanumeric string>
<indicant>::= <alphanumeric string>
<empty>::=
;

    REFERENCE(MODE) PROCEDURE INMODE;
    IF LOOK1("INT")OR LOOK1("REAL")OR LOOK1("BOOL")OR
       LOOK1("CHAR")OR LOOK1("FORMAT") THEN
    (BEGIN REFERENCE(TAG) T;
          T:=INTAG(FALSE);
          ENTER(TAGNM(T)(0|6),0,NULL)
    END)ELSE
    IF LOOK1("LONG") THEN INSHONGS("LONG") ELSE
    IF LOOK1("SHORT") THEN INSHONGS("SHORT") ELSE
    IF LOOK1("REF") THEN
    (BEGIN SEQTAG;
          ENTER("REF",0,PRAM(INMODE,NULL,NULL))
    END)ELSE
    IF LOOK1("(") THEN
    (BEGIN REFERENCE(MODE) RES;
          INTEGER N;
          SEQTAG;
          N:=0;
          WHILE LOOK1(",") DO
          BEGIN N:=N+1;
                SEQTAG
          END;
          IF ¬LOOK1(")") THEN
          BEGIN ERRORFLAG:=TRUE;
                WRITE("***ERROR***  ROWOF NOT TERMINATED ",
                      "BY ')'");
```

```
            IOCONTROL(2)
      END;
      SEQTAG;
      RES:=ENTER("ROWOF",0,PRAM(INMODE,NULL,NULL));
      FOR I:=1 UNTIL N DO
      RES:=ENTER("ROW",0,PRAM(RES,NULL,NULL));
      RES
END) ELSE
IF LOOK1("PROC") THEN
(BEGIN REFERENCE(PRAM) P;
      SEQTAG;
      P:=INPRAM(FALSE,FALSE);
      IF LOOK1("VOID") THEN
      (BEGIN SEQTAG;
            ENTER("PRVOID",0,P)
      END) ELSE
      ENTER("PROC",0,PRAM(INMODE,NULL,P))
END) ELSE
IF LOOK1("UNION") THEN
(BEGIN SEQTAG;
      ENTER("UNION",0,INPRAM(FALSE,TRUE))
END) ELSE
IF LOOK1("STRUCT") THEN
(BEGIN SEQTAG;
      ENTER("STRUCT",0,INPRAM(TRUE,TRUE))
END) ELSE
ENTER("TAG",0,INTAG(TRUE));

REFERENCE(MODE) PROCEDURE ENTER(STRING(6) VALUE KY;
      INTEGER VALUE LN;REFERENCE(PRAM,TAG) VALUE PR);
BEGIN REFERENCE(MODE) M,L;
      INTEGER N;
      M:=MODE(KY,LN,,,0,,PR,,);
      IF KY¬="TAG   " THEN
      IF(BEGIN N:=0;
            WHILE PR¬=NULL DO
            BEGIN N:=N+1;
                  PR:=PRAMLK(PR)
            END;
            MODENF(M):=N;
            IF MODES=NULL THEN TRUE ELSE
            GTPRIO(MODES,M)
        END)THEN
      BEGIN MODEFW(M):=MODES;
            MODES:=M
      END ELSE
      BEGIN L:=MODES;
            WHILE IF MODEFW(L)=NULL THEN FALSE ELSE
                  GTPRIO(M,MODEFW(L)) DO
            L:=MODEFW(L);
            MODEFW(M):=MODEFW(L);
            MODEFW(L):=M
```

```
      END;
      MODEMK(M):=TRUE;
      M
END;

REFERENCE(PRAM) PROCEDURE INPRAM(LOGICAL VALUE T,F);
IF IF ¬LOOK1("(") THEN TRUE ELSE
      LOOK2(",") CR LOOK2(")") THEN
(BEGIN IF F THEN
      BEGIN ERRORFLAG:=TRUE;
            WRITE("***ERROR***  NOPRAM NONPROC")
      END;
      NULL
END) ELSE
(BEGIN REFERENCE(PRAM) P,Q,R;
      P:=NULL;
      WHILE(BEGIN SEQTAG;
                  P:=PRAM(INMODE,NULL,P);
                  IF T THEN PRAMTG(P):=INTAG(TRUE);
                  IF(¬LOOK1(",")) AND(¬LOOK1(")")) THEN
                  (BEGIN ERRORFLAG:=TRUE;
                        WRITE("***ERROR***  BAD DELIMI",
                           "TER IN FIELD");
                        IOCONTROL(2);
                        FALSE
                  END) ELSE
                  LOOK1(",")
            END) DO;
      SEQTAG;
      Q:=NULL;
      WHILE P¬=NULL DO
      BEGIN R:=Q;
            Q:=P;
            P:=PRAMLK(P);
            PRAMLK(Q):=R
      END;
      Q
END);

REFERENCE(MODE) PROCEDURE INSHONGS(STRING(5) VALUE S);
BEGIN INTEGER N;
      SEQTAG;
      N:=1;
      WHILE LOOK1(S) DO
      BEGIN N:=N+1;
            SEQTAG
      END;
      IF LOOK1("INT") OR LOOK1("REAL") THEN
      (BEGIN REFERENCE(TAG) T;
            T:=INTAG(FALSE);
            ENTER(TAGNM(T)(0|6),(IF S="LONG "THEN N ELSE
                  -N),NULL)
```

```
      END)ELSE
      (BEGIN ERRORFLAG:=TRUE;
            WRITE("***ERROR***  SHENGTH OF NON-INTREAL",
                  ">0");
            IOCONTROL(2);
            NULL
      END)
END;

INTEGER AHEAD;
REFERENCE(TAG) NEXTELEM1,NEXTELEM2;

REFERENCE(TAG) PROCEDURE INTAG(LOGICAL VALUE ALPH);
BEGIN REFERENCE(TAG) T;
      LOGICAL X;
      X:=LOOK1(" X");
      T:=NEXTELEM1;
      IF(TAGNM(T)(0|1)<"A")AND(TAGNM(T)(0|1)¬="@")AND
            ALPH THEN
      (BEGIN ERRORFLAG:=TRUE;
            WRITE("***ERROR***  MISSING ALPHA STRING ",
                  "BEFORE '",TAGNM(T)(0|1),"'");
            NULL
      END)ELSE
      (BEGIN
      AHEAD:=AHEAD-1;
      IF AHEAD>0 THEN NEXTELEM1:=NEXTELEM2;
      T
      END)
END;

PROCEDURE SEQTAG;
BEGIN REFERENCE(TAG) T;
      T:=INTAG(FALSE)
END;

LOGICAL PROCEDURE LOOK1(STRING(16) VALUE S);
BEGIN IF AHEAD=0 THEN
      BEGIN AHEAD:=1;
            NEXTELEM1:=INELEM
      END;
      IF NEXTELEM1=NULL THEN S="                " ELSE
      IF TAGLK(NEXTELEM1)¬=NULL THEN FALSE ELSE
      TAGNM(NEXTELEM1)=S
END;

LOGICAL PROCEDURE LOOK2(STRING(16) VALUE S);
BEGIN IF AHEAD=0 THEN
      BEGIN AHEAD:=2;
            NEXTELEM1:=INELEM;
            NEXTELEM2:=INELEM
      END ELSE
```

```
      IF AHEAD=1 THEN
      BEGIN AHEAD:=2;
            NEXTELEM2:=INELEM
      END;
      IF NEXTELEM2=NULL THEN S="                " ELSE
      IF TAGLK(NEXTELEM2)¬=NULL THEN FALSE ELSE
      TAGNM(NEXTELEM2)=S
END;

REFERENCE(TAG) PROCEDURE INELEM;
(BEGIN STRING(1) CHAR;
      STRING(16) GROUP;
      INTEGER P;
      REFERENCE(TAG) RES,POINT;
      WHILE(BEGIN CHAR:=INCHAR;
                  CHAR=" "
            END)DO;
      IF(CHAR<"A")AND(CHAR¬="@") THEN TAG(CHAR,NULL)ELSE
      (BEGIN GROUP:=CHAR;
            P:=1;
            RES:=POINT:=TAG;
            WHILE(BEGIN CHAR:=LOOKCHAR;
                        (CHAR>="A")OR(CHAR="@")
                  END)DO
            IF(BEGIN SEQCHAR;
                     P<16
               END)THEN
            BEGIN GROUP(P|1):=CHAR;
                  P:=P+1;
            END ELSE
            BEGIN TAGNM(POINT):=GROUP;
                  GROUP:=CHAR;
                  P:=1;
                  TAGLK(POINT):=TAG;
                  POINT:=TAGLK(POINT)
            END;
            TAGIK(POINT):=NULL;
      TAGNM(POINT):=GROUP;
            RES
      END)
END);

INTEGER AHEADCHAR,CHARPOINT;
LOGICAL ECHO,CHARQUO;
STRING(1) NEXTCHAR;
STRING(256) CHARBUF;

STRING(1) PROCEDURE INCHAR;
BEGIN STRING(1) CHAR;
      CHAR:=LOOKCHAR;
      SEQCHAR;
      CHAR
```

```
END;

PROCEDURE SEQCHAR;
IF ¬CHARQUO THEN
IF AHEADCHAR¬=0 THEN AHEADCHAR:=AHEADCHAR-1 ELSE
BEGIN STRING(1) C;
      C:=LOOKCHAR;
      AHEADCHAR:=0
END;

STRING(1) PROCEDURE LOOKCHAR;
BEGIN STRING(1) C;
      WHILE(BEGIN C:=LOOKCHARNC;
                  C="#"
            END)DO
      BEGIN AHEADCHAR:=0;
            WHILE(BEGIN C:=LOOKCHARNC;
                        AHEADCHAR:=0;
                        C¬="#"
                  END)DO
      END;
      CHARQUO:=C=";";
      C
END;

STRING(1) PROCEDURE LOOKCHARNC;
IF AHEADCHAR=1 THEN NEXTCHAR ELSE
(BEGIN IF CHARPOINT>255 THEN
      BEGIN READCARD(CHARBUF);
            CHARPOINT:=0;
      IF ECHO THEN
      BEGIN WRITE("***ECHO ***  ",CHARBUF(0|57));
            IOCONTROL(2)
      END
      END;
      NEXTCHAR:=CHARBUF(CHARPOINT|1);
      CHARPOINT:=CHARPOINT+1;
      AHEADCHAR:=1;
      NEXTCHAR
END);

INTEGER N,CLS;
LOGICAL MK;
REFERENCE(TAG) TG;
REFERENCE(MODE) MD;
REFERENCE(TABLE) I;
REFERENCE(PRAM) PR;

WHILE(BEGIN
AHEAD:=0;
AHEADCHAR:=0;
CHARPOINT:=256;
```

```
    NUMBINDIC:=0;
    CHARQUO:=FALSE;
    INDICANTS:=NULL;
    MODES:=NULL;
    ORDERFLAG:=FALSE;
    ECHO:=FALSE;

COMMENT
      The commands supported by the program are
EXIT                 stop the program
RESTART              clear all definitions and restart
                     the program
ECHO                 echo all input to the output
ORDER                list all defined indicants in order
FORM <indicant>      print the canonical form of the
                     mode designated by <indicant>
MODE <indicant>=<mode>    define <indicant>

      All commands are terminated by semicolons.  ;

    WHILE(¬LOOK1("EXIT"))AND(¬LOOK1("RESTART")) DO
    BEGIN IF LOOK1(";") THEN ELSE
          IF LOOK1("ECHO") THEN ECHO:=TRUE ELSE
    IF LOOK1("NOECHO") THEN ECHO:=FALSE ELSE
          IF LOOK1("ORDER") THEN
          BEGIN ORDER;
                N:=0;
                CLS:=1;
                I:=INDICANTS;
                WHILE I¬=NULL DO
                BEGIN IF MODEKY(TABLEMD(I))="TAG   " THEN
                      N:=N+1;
                      I:=TABLELK(I)
                END;
                WHILE N<NUMBINDIC DO
                BEGIN I:=INDICANTS;
                      MK:=FALSE;
                      WHILE I¬=NULL DO
                      IF MODEKY(TABLEMD(I))="TAG   " THEN
                      I:=TABLELK(I) ELSE
                      IF MODECLS(TABLEMD(I))¬=CLS THEN
                      I:=TABLELK(I) ELSE
                      BEGIN IF N=0 THEN ELSE
                            IF MK THEN WRITEON("=") ELSE
                            WRITEON(",");
                            MK:=TRUE;
                            N:=N+1;
                            PRINTTAG(TABLENM(I));
                            I:=TABLELK(I)
                      END;
                      CLS:=CLS+1
                END;
```

```
            WRITEON(";");
            IOCONTROL(2)
      END ELSE
IF LOOK1("FORM") THEN
      BEGIN SEQTAG;
            ERRORFLAG:=FALSE;
            IF ¬ERRORFLAG THEN
            BEGIN
            TG:=INTAG(TRUE);
            ORDER;
            PLACE:=0;
            MD:=MODES;
            WHILE MD¬=NULL DO
            BEGIN MODEPLC(MD):=0;
                  MD:=MODEFW(MD)
            END;
            MD:=LOOKUP(TG);
            WRITE("MODE ");
            PRINTTAG(TG);
            WRITEON("=");
            PRINTMODE(MD);
            WRITEON(";");
            IOCONTROL(2)
            END
      END ELSE
IF LOOK1("MODE") THEN
BEGIN SEQTAG;
            ERRORFLAG:=FALSE;
            TG:=INTAG(TRUE);
            IF ¬ERRORFLAG THEN
            BEGIN
            IF LOOK1("=") THEN SEQTAG ELSE
            BEGIN WRITE("***ERROR*** NO '='");
                  IOCONTROL(2)
            END;
            MD:=MODES;
            WHILE MD¬=NULL DO
            BEGIN MODEMK(MD):=FALSE;
                  MD:=MODEFW(MD)
            END;
            MD:=INMODE;
            IF ERRORFLAG THEN
            BEGIN WHILE IF MODES=NULL THEN FALSE ELSE
                        MODEMK(MODES) DO
                  MODES:=MODEFW(MODES);
                  MD:=MODES;
                  IF MD¬=NULL THEN
                  WHILE MODEFW(MD)¬=NULL DO
                  IF MODEMK(MODEFW(MD)) THEN
                  MODEFW(MD):=MODEFW(MODEFW(MD)) ELSE
                  MD:=MODEFW(MD)
            END ELSE
```

```
                BEGIN I:=INDICANTS;
                      ORDERFLAG:=FALSE;
                      WHILE IF I=NULL THEN FALSE ELSE
                            ¬EQTAG(TG,TABLENM(I)) DO
                      I:=TABLELK(I);
                      IF I=NULL THEN
                      BEGIN NUMBINDIC:=NUMBINDIC+1;
                            INDICANTS:=TABLE(TG,,MD,
                                  INDICANTS)
                      END ELSE
                      TABLEMD(I):=MD
                 END
                 END
           END ELSE
           BEGIN WRITE("***ERROR***  UNRECOGNIZABLE COMMAND");
                 IOCONTROL(2)
           END;
           WHILE ¬LOOK1(";") DO
           SEQTAG;
          CHARQUO:=FALSE;
           AHEADCHAR:=0;
           SEQTAG
    END;
    LOOK1("RESTART")
    END)DO;
    WRITE("***GOOD-BYE***")
END.
```

EXAMPLES

```
# The following examples illustrate the operation of the
  program listed in this appendix.  The ordering produced by
  the program differs from the ordering described by the
  algorithm in Chapter II and from that of the modified form
  of the algorithm described in Chapter IV.  The difference
  is in the ordering of terminals T(Mi).  The canonical form
  produced by the program  differs from that of Section 3.5
  in that primative modes are not included in the "counting"
  process that produces the integers used to represent the
  loops in the graph which represents a mode.
        Comments can be entered in comment-symbols[R1.1.3i],
  namely "##".  Input is represented by lower-case lines,
  and output from the program is represented by upper-case
  lines. #


mode a=union(ref a, ref b, int);
mode b=union(ref a, real);
order;
A,B;
form a;
MODE A=UNION (INT ,REF 1 ,REF UNION (REAL ,2 ));
form b;
MODE B=UNION (REAL ,REF UNICN (INT ,2 ,REF 1 ));
restart;

# Example from Section 11.11 of the Report (also used in
  Section 3.5 of the thesis). #

mode form=union(ref const, ref var, ref triple,ref call);
mode const=struct(real value);
mode var=struct(string name, real value);
mode triple=struct(form leftoperand, int operator,
      form rightoperand);
mode function=struct(ref var boundvar, form body);
mode call=struct(ref function functionname,
      form parameter);
  mode string=()char;
order;
STRING,CONST,FUNCTION,CALL,VAR,TRIPLE,FORM;
form form;
MODE FORM=UNION (REF STRUCT (REAL VALUE),REF STRUCT (REF
```

```
STRUCT (REF STRUCT (()CHAR NAME,REAL VALUE)BOUNDVAR,1 BODY)
FUNCTIONNAME,1 PARAMETER),8 ,REF STRUCT (1 LEFTOPERAND,INT
OPERATOR,1 RIGHTOPERAND));
restart;

# Example to illustrate the order used by the program for
  the terminals T(Mi). #

mode x=union(int,long int,real,long real,bool,char,format,
      ref int,()int,(,)int,()()int,proc void,proc int,
      proc(int)void,proc(int)int,struct(int i));
form x;
MODE X=UNION (BOOL ,CHAR ,FORMAT ,INT ,LONG INT ,PROC INT ,
PROC (INT )INT ,PRCC VOID ,PROC (INT )VOID ,REAL ,LONG REAL
,REF INT ,(,)INT ,()INT ,()8 ,STRUCT (INT I));
restart;

#Example to illustrate certain properties of united modes. #

mode a=union(int,real);
mode b=union(real,int);
mode c=union(real,union(int,real));
order;
C=B=A;
form a;
MODE A=UNION (INT ,REAL );
form b;
MODE B=UNION (INT ,REAL );
exit;
***GOOD-BYE***
```

APPENDIX B

The algorithm of Chapter II is illustrated in this appendix by an example using the following mode-declarations:

```
mode a = union(ref b,ref a, proc(bool)char);
mode b = union(ref a, d);
mode c = proc(int)void;
mode d = [1:2,1:3]int;
mode e = struct(d x,bool y);
mode f = struct(d x, char y);
mode g = struct(d x, bool z);
mode h = union(ref a, d, proc(bool, char)d);
```

In the example the Mi are, before elimination of indicants:

```
M0  = union(M1, M3, M5)
M1  = ref M2
M2  = b
M3  = ref M4
M4  = a
M5  = proc(M6)M7
M6  = union(M7, M9)
M7  = ref M8
M8  = a
M9  = d
M10 = proc(M11)void
M11 = int
M12 = rowof M13
M13 = rowof M14
M14 = int
M15 = struct(M16 x, M17 y)
M16 = d
M17 = bool
M18 = struct(M19 x, M20 y)
M19 = d
M20 = char
M21 = struct(M22 x, M23 z)
M22 = d
M23 = char
M24 = union(M25, M27, M28)
M25 = ref M26
M26 = a
M27 = d
M28 = proc(M29, M30)M31
M29 = bool
M30 = char
```

M31 = d .

After elimination of indicants, the modified modes are:

M1 = ref M6
M3 = ref M0
M6 = union(M7, M12)
M7 = ref M0
M15 = struct(M12 x, M17 y)
M18 = struct(M12 x, M20 y)
M21 = struct(M12 x, M23 z)
M24 = union(M25, M12, M28)
M25 = ref M0
M28 = proc(M29, M30)M31 .

Step 1. The classes defined in substep (a) are

(i) C(M15) = C(M18) = 1
C(M11) = 2

(ii) C(M0) = C(M6) = C(M24) = 3

(iii)
C(M17) = C(M29) = 4
C(M20) = C(M23) = C(M30) = 5
C(M1) = C(M3) = C(M7) = C(M25) = 6
C(M12) = C(m13) = 7
C(M10) = 8
C(M5) = 9
C(M28) = 10
C(M11) = C(M14) = 11.

The value of P is therefore 11.

Step 2. At the first iteration of the algorithm:

for M0: L(M0) = 2
S(M0,1) = 6
S(M0,2) = 9

for M6: L(M6) = 2
S(M6,1) = 6
S(M6,2) = 7

for M24: L(M24) = 3
S(M24,1) = 6
S(M24,2) = 7
S(M24,3) = 10.

Step 3. The first class to be divided on its constituents is that for which C(Mi) = 1. Its members are M15 and M18.

CONST(M15,1) = M12
CONST(M18,1) = M12

so C(CONST(M15,1)) = C(CONST(M18,1)) = 7.

CONST(M15,2) = M17
CONST(M18,2) = M20

so C(CONST(M15,2)) = 4 < 5 = C(CONST(M18,2)).

Therefore, using n = 2 (for substep (b)), the step redefines C(M18) = 2, and each Mi for which C(Mi) > 1 has its class number incremented by one.

Returning to step 2 produces no change in the values of the S(Mi, k), except to reflect the increase in the class-number values. On the second iteration, the class for which C(Mi) = 8 is divided. Its members are M12 and M13.

CONST(M12,1) = M13
CONST(M13,1) = M14
so C(CONST(M12,1)) = 8 < 12 = C(CONST(M13,1)).

Therefore, using n = 1, the step redefines C(M13) = 9, and leaves the classes as follows:

C(M15) = 1
C(M18) = 2
C(M21) = 3
C(M0) = C(M6) = C(M24) = 4
C(M17) = C(M29) = 5
C(M20) = C(M23) = C(M30) = 6
C(M1) = C(M3) = C(M7) = C(M26) = 7
C(M12) = 8
C(M13) = 9
C(M10) = 10
C(M5) = 11
C(M28) = 12
C(M11) = C(M14) = 13.

Returning to step 2 produces no changes and step 3 now fails. In step 4, however, for the class containing M0, M6 and M24,

L(M0) = L(M6) = 2
L(M24) = 3.

Therefore, step 4 redefines C(M4) = 5, and likewise, all class numbers above 4 are incremented by one.

Step 2 again produces no changes, and steps 3 and 4 fail to produce any results. At step 5, the class containing M0 and M6 is divided:

S(M0,1) = S(M6,1) = 7
S(M6,2) = 11 < 12 = S(M0,2)

so that the united modes M0 and M6 are divided and ordered, not on their actually distinct first constituents, nor on their actually different lengths, but on their last fields. The only remaining division to be made is to divide the class of references containing M1, M3, M7 and M25. This division leaves the terminal set of classes:

C(M0) = 5
C(M1) = 9
C(M3) = C(M7) = 10
C(M5) = 14
C(M6) = 4
C(M10) = 13
C(M11) = C(M14) = 15
C(M12) = 11
C(M13) = 12
C(M15) = 1
C(M17) = C(M29) = 7
C(M18) = 2
C(M20) = C(M23) = C(M30) = 8
C(M21) = 3
C(M24) = 6
C(M28) = 15