OPERATOR IDENTIFICATION IN ALGOL 68

by

YING KWAN

B.Sc., Taiwan Normal University, 1963

M.A., University of British Columbia, 1970

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

COMPUTER SCIENCE

We accept this thesis as conforming to the

required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April, 1973

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver 8, Canada

Date 16th April, 1973.

ABSTRACT

The special feature that modes and operators can be defined by the user of ALGOL 68 has induced the problems of coercion, balancing and operator identification.

This work deals with the mode manipulation and operator identification in ALGOL 68. The algorithms are based on those of [Z]. Some of the revisions to the ALGOL 68 Report concerning modes, such as no proceduring, the void symbol, the definition of NONPROC, the definition of a vacuum, and the hipping of a vacuum are included. The program in ALGOL W is based on that of [P].

The program described in this thesis does four main jobs: mode equivalencing, mode coercion, mode balancing and operator identification. In mode equivalencing, it checks the context conditions concerning "showing" [R.4.4.4] and the multiple occurrence of the same field selector in a structure [R.4.4.3e], and checks related modes in unions [R.4.4.3b,d]. In mode coercion, it determines the coercion steps. This is also a basic part of mode balancing and operator identification. In balancing, it also considers collateral displays. This is a model for the operator identification part of an ALGOL 68 compiler.

ACKNOWLEDGEMENTS

I am deeply indebted to Professor J. E. L. Peck for initiating my study of ALGOL 68 and ALGOL W, for suggesting the topic of this thesis and for rendering invaluable assistance, encouragement and patience throughout the course of my work. I would like to thank the group which is working under Dr. Peck for the ALGOL 68 implementation, Dr. W. J. Hansen and especially Dr. M. Zosel for many helpful suggestions and discussions.

I gratefully acknowledge the financial support of NRC.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

A very important factor of the power of a general purpose language is the facility incorporated to treat the bits of computer memory as different data types: bits, logical values, characters, integers, reals, complex numbers, arrays, strings, lists, trees, etc. Some programming languages allow the user to define new data types and to extensively use the defined data types by:

(1) using them in further definitions,

(2) using them as parameters to procedures,

(3) extending the range of existing operators to include them and

(4) define new operators to operate on them. ALGOL 68 permits the user to define data types termed modes, and allows the four features mentioned. This generality in the language poses the following problems for the compiler implementor:

(1) when there are two modes representing the same data type (equivalencing),

(2) when and how a mode can be transformed to another (coercion),

(3) to what mode should elements of a list of modes be coerced (balancing) and

(4) how the operator definition in an applied occurrence of the operator is determined (operator identification).

Example(1), real x := 3; where the value possessed by 3 is of mode integral and the value which is to be assigned must be of mode real. So the value of mode real which is equivalent to 3 must be derived before the assigment. The process of transforming integral to real is a coercion called widening.

Example(2), real x,y; x:= x+y; in the expression x+y, the operator is +, the operands are x and y which are of mode reference to real. This is an applied occurrence of the operator +. The defining occurrence of the operator + must be determined. In the Report, there are at least 16 defining occurrences of the operator + [R.10.2.3i,j, 10.2.4i,j, 10.2.5a,b, 10.2.6b, 10.2.7j,k,p,q,r,s, 10.2.10j,k,l.]. Among them,

op + = (L real a,b) L real : a--b; [R.10.2.4i] is the only defining occurrence to be identified by the formula x+y.

Coercion is the way through which one mode can be transformed to another. Only COERCENDS can be coerced. There are five different positions for a COERCEND: strong, firm, meek, weak and soft. There are seven different coercions: deproceduring, dereferencing, uniting, rowing, widening, hipping and voiding.

Balancing occurs in a conditional clause, serial clause, the firm MODE balance PACK in a firm collateral row of MODE clause or the strong structure PACK in a strong

collateral structure clause. There are also five positions of balancing. In a FEAT balance, one of the constituent CLAUSE must be FEAT, while the other may be strong.

Operator identification is necessary when there is more than one defining occurrence of an operator. In operator-identification, the position of an operand is always firm. In this process of mode manipulation and operator identification, the problem of determining whether (a|x|y)(i) is a slice or a call can be solved when the (a priori) modes of x and y are given.

This work deals with the mode manipulation and operator identification in ALGOL 68 with the algorithms based on those of [Z]. It is based on the revised syntax which is given in Appendix B so some of the revisions to the ALGOL 68 Report concerning modes, such as nc proceduring, the void symbol, the definition of NONPROC, the definition of a vacuum, and the hipping of a vacuum are included. The program in ALGOL W is based on that of [P].

The program described in this thesis does four main jobs: mode equivalencing, mode coercion, mode balancing and operator identification. In mode equivalencing, it checks the context conditions concerning "showing" [R.4.4.4] and the multiple occurrence of the same field selector in a structure [R.4.4.3e], and checks related modes in unions [R.4.4.3b,d]. In mode coercion, it determines the coercion steps. This is also a basic part of mode balancing and

operator identification. In balancing, it also considers collateral displays.

This work is an extension of [P], so the input, output format and the internal representation are the same and it is different from [P] in the following respects:

(1) An echo print for input is added.

(2) An explicit void symbol is included.

(3) The treatment of skips, nihils, jumps and vacuums are included and they are considered as special modes.

(4) The consideration starts from the a priori mode (the source mode) that avoids some unnecessary flagging of some modes.

(5) Makes use of vectors of bitstrings and the operations AND and OR instead of the APL operator A/B in many places. The length of the parameterlist remains the same throughout the process here while that is not the case in [P].

(6) Changes in the revised Report such as no proceduring, meek coercion are included and

(7) The identification of a(i) as slice or call is also included.

In this thesis, each chapter that describes a part of the program, is divided into three sections which may be subdivided. In the first section, the meanings of the terms concerned in that chapter are shown, and perhaps some syntax rules about them are given. So the terms used in the above

paragraphs are not explained here. In the second section, the algorithm is given and in the third section, an example is given to show how the program works.

For the sake of brevity, contractions such as int for integral, ref for reference to, etc., are used throughout this work.

CHAPTER II

MODE REPRESENTATION

This chapter describes how a mode is declared and how it is stored.

2.1. Mode declarers:

The syntax of declarers is provided in the Report: [R.1.2.1], [R.1.2.2], [R.1.2.7], [R.7.1.1] and [R.7.2.1]. Examples of mode declarations are:

```
mode md = struct ( string t, cw, bits flag, int mn, nf, ref
          union (nd, md) link, field);
mode nd = struct ( ref md v, ref nd nxtmd).
```

2.2. The mode storage grammar:

Modes can be represented by a grammar [Z, P2]. Let the primitive modes be P1 = { void, bool, int, real, char, format, bits, bytes, skip, jump, nil, vacuum} (void, skip, jump, nil and vacuum are not really modes, however, they are treated like modes. The long versions of int, real and compl are not considered in this thesis). Let the prefixes be subdivided into two sets :

P2 = { ref, proc, row, rowof } and

P3 = { union, struct, procp } where procp is procedure with parameters (row, rowof are used for [ ]).

Let the MODE declarers be viewed as a grammar g = < T,N,P > where

the set of terminals is

T = P1 U P2 U P3 U P4,

where P1,P2,P3 are defined above. P4 is the set of all field selectors of the defined structures.

N is the set of nonterminals, i.e., the set of all the modes (each represented by an integer) and

P is the set of production rules. A production rule is in the form of

mx = p my(1) my(2) ... my(n)

with mx, my(1), my(2), ..., my(n) in N, p called the terminal, in T;

if p is in P1 then my(1) ... my(n) is empty,

if p is in P2 then n=1 and my(1) is the declarer following p in mx,

if p is in P4 then n=1 and my(1) is the declarer of the portrayal containing the field selector p,

if p is in P3 then n≥1 and the my(i)'s for i = 1,...,n are the constituents of p where

if p is "union", my(1), my(2), ... my(n) are the constituent modes for p;

if p is "struct" then my(i) is the i-th portrayal;

if p is procp then my(1),...my(n-1) are the modes of the parameters and my(n) is the mode delivered by the procedure.

Portrayals of a structure [revised R.7.1.1.ea] are part of

the mode so there are modes in the form mx = q my with q not in the union of P1, P2 and P3. In this case, q must be a field selector of a structure and my is the mode of q. Note that it is convenient here to interchange the order in a portrayal.

As modes can be infinite in length but not the declarers, modes are stored as declarers. Each mode declarer is represented by a number. The primitive modes considered in this study are:

m0 = void,

m1 = bool,

m2 = int,

m3 = real,

m4 = char,

m5 = format,

m6 = bits,

m7 = bytes,

m12 = skip,

m13 = jump,

m14 = nil,

m15 = vacuum.

The standard modes which reside in the program are the primitives together with the following:

m8 = struct m10 m11 (complex)

m9 = rowof m4 (string)

m10 = re m3

m11 = im m3.

Here m10 and m11 are the two portrayals of m8.

2.3.1. Representation of the modes in the program:

As the program is written in ALGOL W, all the terms used here are in ALGOL W sense.

In the program, a record 'md' is allocated to each mode number defined in the above way. The record class 'md' is defined as

record md (string (6) t, cw; bits flag; integer mn, nf;
reference (md, nd) link, field);

t	cw	flag	mn	nf	link	field
" "	" "	#	int	int	--> (nd,md)	--> (nd,md)

where nd is another record class defined as

record nd (reference (md) v; reference (nd) nxtmd);

v	nxtmd
--> (md)	--> (nd)

which is used for a list of modes. (The arrow indicates a reference.)

In the record 'md', the first field 't' is a string of 6 characters, which is the terminal of the mode. The second field 'cw' is a string of 6 characters, which is used as temporary storage for the coercion word for the mode in the

process of coercion. The third field 'flag' is a bitstring used to store the flags in the process of coercion. This is the most useful field in the manipulation. The fourth field 'mn' is an integer for mode number. This is the integer of the left hand side of a rule of the mode grammar, or a fixed number for a standard mode. For example: 'mn' of void is 0, bool is 1. The fifth field 'nf' is an integer to show the number of modes referenced by 'field' of the mode. If the terminal is a primitive mode then the 'nf' is 0, otherwise it is equal to the number of modes in the mode list to follow the terminal in the rule. The sixth field 'link' is a reference to either an 'md' or an 'nd'. 'link' is used as working storage to reference an 'md' to link those modes which are of the same class in equivalence or to link backwards in coercion. The seventh field 'field' is also a reference to either an 'md' or an 'nd'. For a primitive mode, it is null otherwise it points to the mode list that contains the constituents of the terminal, if the terminal is union, struct or procp, or it points to the mode that follows the terminal of the present mode.

A record 'md' is also allocated to something that is not really a mode in the ALGOL 68 sense, but that is looked upon as a mode and is used to help simplify the manipulation in the program; in this case, the record 'md' is said to be used for a 'pseudo-mode'. There are two cases of a 'pseudo-mode':

(1) the mode0 in the program:

" "	" "	#0	-1	0	null	null

(2) An 'md' whose 'cw'-field contains "cond", "seri" or "coll" to show that its 'link'-field points to a mode list for a conditional, serial or collateral clause. An example is:

" "	"cond"	#0	-1	0	-->(nd)	null

2.3.2. An example to show how it works in the program:

In using the program, mode declarers are entered by using the command "GRAMMAR" and a set of rules of grammar to be entered followed by an integer less than -1 that shows the end of the set of rules. Each rule of the mode grammar is entered as

m t p

where (a) m is an integer such that $15 < m < 35$. A warning will be given and the rule will be entered at the end of the whole set of rules for the first time this restriction is broken, if it is broken more than once, only the last of the rules breaking this restriction will be entered and the others that break this restriction will be neglected.

(b) t is a terminal which is a string so that it must be enclosed in quotes.

(c) p is either a mode i.e. an integer i such that $0 \leq i \leq 35$ or a mode list i.e. a sequence of modes

followed by an integer less than -1.

Example:

```
"grammar"
16   "ref"   17
17   "union" 2 18 19 -2
18   "rowof" 3
19   "struct" 20 21 23 -2
20   "a"   9
21   "b"   22
22   "ref"   19
23   "c"   22
24   "ref"   18
25   "proc"   3
-2
```

In the above example,

m16 = ref union (int, [ ] real, m19),

m19 = struct (string a, ref m19 b, ref m19 c),

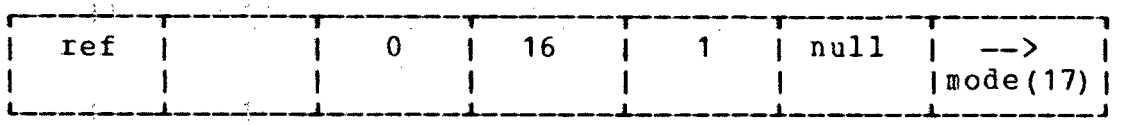
m18 = rowof real,

m24 = ref [ ] real,

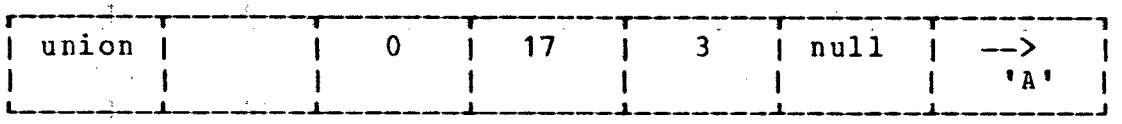
m25 = proc real.

The -2 in m17 is to terminate the list of modes in the union, the -2 in m19 is to terminate the list of modes in the fields of the structure. The last -2 is to terminate the rules of the grammar. The command "GRAMMAR" then comes to an end. All the modes defined are put in an array of 'md's, MODE, of which the first 16 are fixed. Now some of the elements of MODE are as follows:

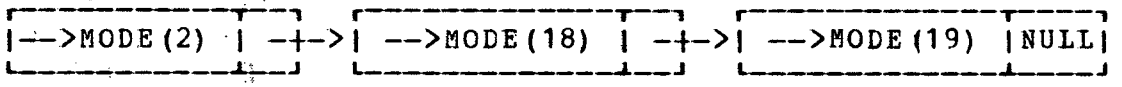
MODE(16) is



MODE(17) is

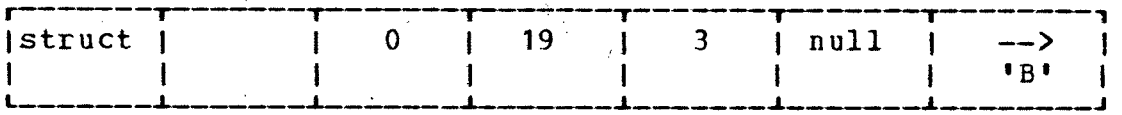


where A is

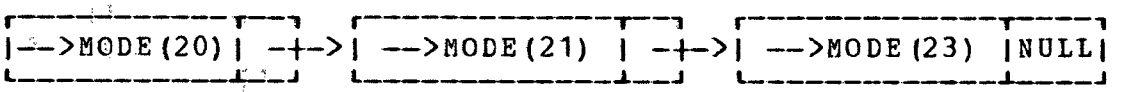


MODE(18) is

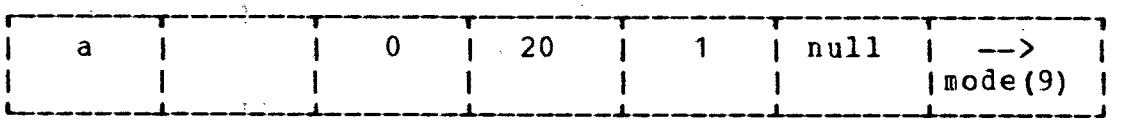
MODE(19) is



where B is



MODE(20) is



and so on.

CHAPTER III

MODE EQUIVALENCING

3.1.1. Equivalent modes:

In an ALGOL 68 program, the programmer might have defined some modes which are equivalent, that is they are the same terminal production of the metanotion MODE. An example of equivalent mode declarers:

```
mode u = struct (int i, ref u j);
mode v = struct (int i, ref struct (int i, ref v j) j).
```

3.1.2. Related modes and raveling of a union mode:

Two modes are said to be 'related' to one another if they are both firmly coerceable from one same mode [R.4.4.3.b]. Thus a mode is related to itself. The modes ref proc real and real are related while the modes ref real and proc real are not (the latter two were related in R!), since there is no proceduring coercion (in the revised Report). It is not allowed to define a mode united from two modes related to one another [R4.4.3d]. The relation 'related' is only checked when the modes are the constituent modes of a union. Since the union modes inside a union are raveled, no unions are involved in this process.

If a union mode m1 is a constituent of another union mode m, m1 is to be raveled such that m1 is no longer a constituent of m, but each constituent of m1 is a

constituent of m. Example: union (bool , union (int , char)) will be changed to union (bool , int , char) when its constituents are raveled. Before the relation 'related' is checked, the constituents of the union mode are raveled, sorted in order by mode numbers and duplicated constituents are removed.

3.1.3. Some context conditions:

Since the programmer is allowed to define his own modes and infinite modes may be declared, some conditions must be satisfied such that (1) no mode may allow ambiguous parsing of the program and (2) a value of any mode must not take infinite storage space in the computer.

An example for (1):

mode t = struct (int a, real a).

Examples for (2):

mode f = struct ([1:10] f a, int s).

mode t = ref t .

The first condition is that no duplicate selector is allowed in a structure mode [R.4.4.3.e] and the second is the context condition shielding [R.4.4.4]. The following indications are shielded and so are legal:

(1) mode a = struct (int b, ref a c) where the second occurrence of a is a virtual-declarer following a reference-to-symbol in a portrayal.

(2) mode a = ref struct (int b, a c) where the second occurrence of a is a virtual-declarer contained in a

portrayal contained in a virtual-declarer following a reference-to-symbol.

(3) mode a = procp (a) a where the second occurrence of a is a virtual parameter and the third occurrence of a is a virtual declarer following a virtual-parameters-pack.

A declarer is "showing" if it contains a mode-indication which is not "shielded" (see examples for (2) above).

3.2.1. Definition of equivalent modes in the mode-grammar:

In the mode-grammar described in CHAPTER II, two modes are unequal if the integers representing them are not the same. There is a unique production rule for each nonterminal. In the following, the terminals and the nonterminals are those on the right hand sides of the production rules for the modes. Two modes are not equivalent if and only if one of the following is true:

(1) the terminals are not the same,

(2) if the terminals are not 'union', the numbers of nonterminals are not equal,

(3) if the terminals are not 'union', a nonterminal of one production rule is not equivalent to the corresponding nonterminal of the other,

(4) if the terminals are 'union', there exists a nonterminal of one rule not equivalent to any of the nonterminals of the other.

3.2.2. Algorithms:

In order to save storage space and not to complicate the process of coercion and balancing, modes are equivalenced, that is if two or more modes are equivalent, only one is kept, as soon after they have been entered as possible. Before equivalencing, the context conditions described above are checked and unions are raveled.

The algorithm to check the context conditions of a mode:

Input: mode: a mode whose context conditions are to be checked.

ref: a logical value which is false when the procedure is first called. This is true, if 'ref' has appeared before.

struct: a logical value which is also false when this procedure is first called. This is true, if 'struct' has appeared before.

Output: error message, if the context conditions are not met.

Function: to change duplicated field selectors to "%". To change modes which are "showing" to 'bool'.

Step1: let m be 'mode'. If the flag-field of m is #1, then goto step6.

Step2: set the flag-field of m to #1.

Step3: if the terminal of m is "procp" then goto step5. If the terminal is "ref" then if 'struct' is true then goto step5, if false then set 'ref' true.

If the terminal is "struct" then go to step7.

Step4: if the field-field of m points to an 'md' then recursively check the conditions of the mode pointed to by the field-field of m with the present logical values of 'ref' and 'struct' (by going through step1).

If the field-field of m points to an 'nd' then recursively check the conditions of each mode in the mode list pointed to by the field-field of m with the present logical values of 'ref' and 'struct'.

Step5: set the flag-field of m back to #0. Stop.

Step6: the mode m is showing itself. Set m to 'bcol' to avoid this dangerous situation and a message is given. Stop.

Step7: check for repeated field-selector: check the modes in the mode list pointed to by the field-field of m, if there are two identical terminals, give an error message and set the second one to "%".

Step8: if 'ref' is true then goto step5.

Step9: set 'struct' true. Goto step4.

End of the algorithm for checking context conditions of a mode.

The algorithm for raveling the unions in a mode list which is the constituents of a union:

Input: mdlist: the list of modes which are the constituents of a union mode, to be raveled.

Output: .

Function: to change 'mdlist' to a list which contains the non-union modes of 'mdlist' and constituents of those raveled union modes of 'mdlist'.

Step1: if 'mdlist' is null then goto step2 otherwise do step1.a through step1.g.

Step1.a: if the terminal of v('mdlist') is not "union" then goto step1.f.

Step1.b: let q be a copy of the list of modes pointed to by the field-field of v('mdlist').

Step1.c: let nxtmd('mdlist') be concatenated to q.

Step1.d: set 'mdlist' = q; ravel('mdlist').

Step1.e: goto step2.

Step1.f: let p be equal to nxtmd('mdlist').

Step1.g: ravel(p).

Step2: stop.

End of the algorithm for raveling unions.

The algorithm for equivalencing of all the defined modes:

Input: .

Output: .

Function: to partition all the rules defined in the grammar to equivalence classes such that two rules are in the same class if and only if they are equivalent. For each equivalent class of modes, only one is kept, the rest is discarded.

Step1: partition all the rules defined in the grammar to

equivalence classes such that two rules are in the same class if and only if their terminals on the right hand side are the same.

Step2: partition each class except the class with the terminal "union", obtained in step1 into subclasses, such that two rules are in the same subclass if and only if they have the same number of nonterminals on the right side. So in this new partition of the rules, all rules whose terminals are "union" are in the same class, any other two rules are in the same class if and only if they have the same terminals and equal numbers of nonterminals.

Step3: subdivide each class whose terminal is not "union", in the previous partition into subclasses such that two rules are in the same subclass if and only if the i-th non-terminal of one rule is in the same class as the i-th nonterminal of the other for all $1 \leq i \leq n$ where n is the number of nonterminals on the right hand side of the rule.

Subdivide each class whose terminal is "union" in the previous partition into subclasses such that two rules are in the same subclass if and only if each of the nonterminals of the one is in the same class as one of the nonterminals of the other.

Step4: if no subdivision occurred in step3 then stop otherwise goto step3.

End of the algorithm for equivalencing.

The algorithm to check if there are related modes in a list of modes is as follows:

Input: a list of modes.

Output: a sublist of the input list that contains all the modes such that each of them is related to some other mode in the list (this list is null when no two of the given modes are related).

Step1: mark each mode m1, m2, ..., mk of the list.

Step2: let 'return' be a null mode list.

Step3: for i:=1 until k do step3.a to step3.c.

Step3.a: let m = mi.

Step3.b: if m = proc n or m = ref n then

in case that n is marked, then n and mi are related and search-insert them to 'return' (that means if they were not there then insert them in order) else set m = n and goto step3.b; otherwise,

step3.c: mi has been completely processed.

Step4: 'return' is the mode list to be returned.

End of the algorithm for checking related modes.

3.2.3. Reverse raveling of unions:

The job of 'coercion' is to check if a given mode, the source mode can be transformed to some other mode, the target mode. The source mode is called the a priori mode and the target mode the a posteriori mode. In order to simplify the processes of 'coercion', 'balancing' and 'collateral', a procedure called 'reverse_ravel' is called

immediately after equivalencing. In 'reverse_ravel', defined sub-unions of a union mode are added as fields to the given union mode. For example: if the mode union (int, real, compl) and union (int, real) are defined, union (int, real, compl) will be changed to union (int, real, compl, union (int, real)) by 'reverse_ravel'. In the process of 'coercion', 'balancing' or 'collateral', the a posteriori mode is flagged and the modes from which it can be coerced are also flagged, for example, when the a posteriori mode is real and the position is strong then real and int are both flagged because int can be strongly widened to real. So in the above example, whenever union (int, real, compl) is given as the a posteriori mode in strong or firm coercion, as a balanced mode in balancing or as the mode of an operand in a defined occurrence of an operator which is to be identified, then the modes union (int, real, compl), int, real, compl and union (int, real) should all be flagged. Without 'reverse_raveling', from the mode union (int, real, comp), the mode union (int, real) can be reached only by searching through all the defined modes, and a search for each occurrence of the given union mode as an a posteriori mode is necessary. With 'reverse_raveling', the subunion modes can be accessed from the constituents, and the search is once for all.

The algorithm for reverse raveling a union mode:

Input: mi, a union mode .

Output: 'mi'.

Function: to include the subunions of 'mi' in the list referenced by the 'field'-field of 'mi'.

Step1: for each defined union mode mj other than mi do step2.

Step2: if the mode list pointed to by the field-field of mj is a sublist of the mode list pointed to by the field-field of mi then add mj to the mode list pointed to by the field-field of mi.

Step3: stop.

End of the algorithm for reverse raveling a union mode.

3.3.1. An example to show how to check the context conditions for a given mode:

Suppose the following mode grammar is entered:

```
16   "struct"   19
17   "ref"   16
18   "struct"   20
19   "i"   17
20   "j"   18
-2.
```

That is m16 = struct (ref m16 i) and

m18 = struct (m18 j).

In checking the conditions for m16 :

(1) MODE(16) is checked with 'ref' = false, 'struct' = false: flag(MODE(16)) is set to #1 and the field selectors are checked.

(2) MODE(19) is checked with 'ref' = false, 'struct' = true:

flag(MODE(19)) is set to #1.

(3) MODE(17) is checked with 'ref' = false, 'struct' = true: flag(MODE(17)) is set to #1.

(4) since its terminal is "ref", then flag(MODE(17))is set back to #0, and so are the flag-fields of MODE(19), MODE(16).

In checking the conditions for m18 :

(1) MODE(18) is checked with 'ref' = false, 'struct' = false: flag(MODE(18)) is set to #1.

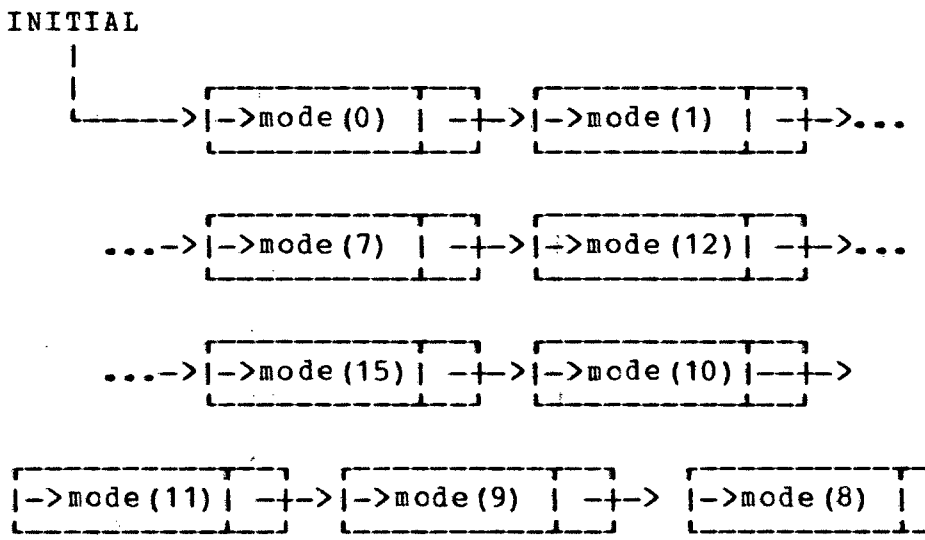
(2) MODE(20) is checked with 'ref' = false, 'struct' = true: flag(MODE(20)) is set to #1

(3) MODE(18) is checked again with 'ref' = false, 'struct' = true: flag(MODE(18)) is #1 already, so an error message is given and MODE(18) is set to 'bool', a mode equivalent to MODE(1).

3.3.2. The mechanism used to equivalence modes in the program:

In the program, an 'nd' INITIAL is used such that each mode in the mode list is from a different class. There is a one to one correspondence of an element of the list INITIAL and a class of the partition of the rules. All the modes in each class are linked together through their link-fields to form a linked list and this list hangs down from that element for this class in the list INITIAL. The flag-field of the modes are used for the class number of the class which the mode is in.

At first, modes are classified by their terminals, i.e. each element of the list INITIAL hangs a list of modes, linked through their link-fields, with the same terminal. The first 12 classes (from 0 to 11th) are for the primitive modes, the 12th and the 13th are for 're' and 'im' of 'complex'. The 14th is for 'rowof' and the 15th is for 'struct'. Classes with terminals other than the above 16, are attached to the end of the list INITIAL. The modes are like this:



The first 15 elements of the list are the standard modes. All modes with the same terminal are in a list hanging down from an element of the list INITIAL, and the class number of each mode is recorded in the flag field. The process of equivalencing is then divided into 'pass'es. In each 'pass' two pointers to mode lists, 'p' and 'final' are initialized to point to the 13-th and the last element of the list INITIAL, and a pointer to a mode list or a mode, 'tail', initialised as a pointer to the last element of

INITIAL, is used to point to the place where modes belonging to a new class should be placed; and a pass is complete when 'p' goes from the 13-th to the end of the list INITIAL. Each 'pass' is divided into 'scans'. A 'scan' for an element m of the mode list INITIAL is a moving of all those modes which were in the same class with m, but are not now, to the place pointed to by TAIL; that is the linked list hanging from m is checked and all those modes in this list that are no longer in the same class as m are grouped together to form a new class hanging down from an element added to the end of the mode list INITIAL.

In 'pass' 1, each class, except the one with terminal "union" is subdivided according to the nf-field. This is step2 of the algorithm. After 'pass' 1, all the union modes are in the same class and each of the other classes contains all the modes which are of the same terminal and have equal numbers of nonterminals. This 'pass' subdivides the classes hanging from elements after the 12-th of the list INITIAL. The class number recorded in the flag-field is updated.

In 'pass' 2, each class is subdivided as in step3 of the algorithm by using the flag-field(s) of the mode(s) pointed to by the field-field of the mode. This 'pass' is repeated until no more new class is formed.

Now the mode list INITIAL contains as many elements as equivalent classes. The link field of each mode is set to point to the first element of its class and all the

field-fields of the modes are changed accordingly, that is, a mode is replaced by the first mode of its class. Only the first element of a class is kept. Discard all duplicated modes and compact the grammar.

For a union mode, the nonterminals are sorted and repeated modes are removed and the context condition of related modes is checked. The detail of how to check if a list of modes contains related modes will be in next section.

In the program, this is initiated by the command "EQUIVALENCE" and no other data are required.

3.3.3. Examples to show how related modes are checked:

In checking the context condition of related modes, the flag-fields of the modes are used for marking the modes. The command to test whether a list of modes contains some related modes is "RELATED" followed by a list of modes. Note that if two modes in the given list are related by involving the coercion 'unite' cannot be detected by this command as all the modes inside a union mode are supposed to have been raveled, and this condition is checked only when the list of modes forms the constituents of a union mode.

Example: suppose the grammar entered is

```
16   "ref"   17
17   "proc"   3
```

```
18   "proc"  19
19   "ref"    3
20   "ref"   19
21   "ref"    2
-2.
```

When the command

"RELATED" 16 18 20 -2

is entered, then

(1) The flag-fields of MODE(16), MODE(18) and MODE(20) are set #1.

(2) Consider ref proc real (MODE(16)), it begins with ref, then check if the flag-field of proc real (i.e. MODE(17)) is #1, it is not.

(3) Consider proc real, it begins with proc then check if the flag-field of real is #1, it is not.

(4) Consider proc ref real (MODE(18)), it begins with proc then check if the flag-field of ref real is #1; it is not.

(5) Consider ref real, it begins with ref then check if the flag-field of real is #1; it is not.

(6) Consider ref ref real (MODE(20)), it begins with ref, then check if the flag-field of ref real is #1; it is not.

(7) Consider ref real, it begins with ref, then check if the flag-field of real is #1; it is not.

This list does not contain any set of related modes.

When the command is

"RELATED" 16 18 20 3 -2

then

(1) The flag-fields of MODE(16), MODE(18), MODE(20) and MODE(3) are set #1.

(2) Consider ref proc real, it begins with ref then check if the flag-field of proc real is #1; it is not.

(3) Consider proc real, it begins with proc, then check if the flag-field of real is #1; it is.

(4) Both real and ref proc real are inserted to the related mode list 'return'.

(5) Consider proc ref real as before and the flag-field of real is #1; so proc ref real is inserted to 'return' (real is there already).

(6) Consider ref ref real as before and the flag-field of real is #1; so ref ref real is also inserted to 'return'.

The mode list 'return' containing real, ref proc real, proc ref real and ref ref real is delivered.

CHAPTER IV

COERCION OF MODES

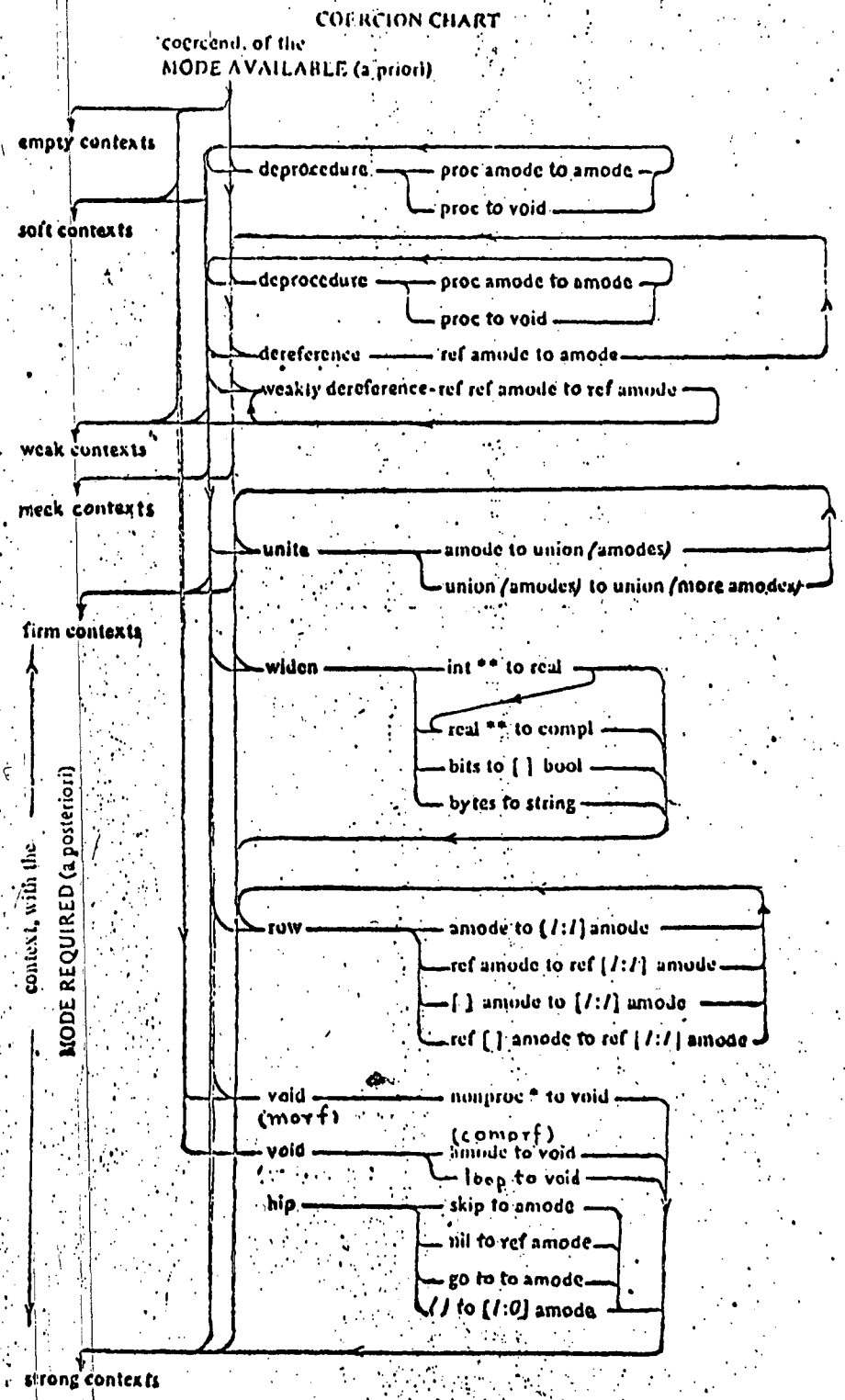
4.1. The coercion process:

In ALGOL 68, since the programmer is allowed to define modes according to the syntax rules [R.7.1.1 and R.7.2.1], the coercion process, that is the way in which a mode can be transformed to another, is more complicated than any other language which does not allow the programmer to have this freedom.

There are five positions for coercion: strong, firm, meek, weak and soft. The position depends on the context. The coercion process is described in the syntax of the language [R.8]. A mode MS is said to be coerceable from a mode MR if there is a path in the syntax such that

SORTETY MS base *==> MR base.

Graphically, the coercion routes described by the syntax are shown in the following chart which is modified from p.208 of [L]. This is for the revised syntax.



* nonproc is all modes except proc moid or refs proc moid where moid stands for amode or void.

** The corresponding longs versions can also be widened.

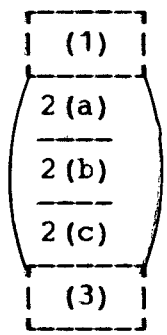
Because the language allows the [ ] symbols to be replaced by () symbols, a problem is posed. For example:

y:=a(i)

where whether a(i) is a slice or a call is not known, because identifier identification is made at the same time as the coercion process. The question is whether the position of the 'a' is soft or weak. In such a case, the a posteriori mode and the sort are not known, however this problem can be solved and its solution included in the coercion process.

4.2. The algorithm for coercion:

The following is the graphical representation of the main theme of the algorithm. Mr is the a priori mode, ms is the a posteriori mode. A state of the state diagram is as follows:



with

(1) the coercion step

(2) the conditions for entry to this state:

(a) the form of the mode under consideration,

(b) the form of mr and

(c) coercend and

(3) the mode to consider next, specified in terms of 2(a).

Note that the coercion step can be taken only when the conditions in (2) are satisfied.

In the graph, if mr is skip, jump, nil or vacuum do the step hipped to see if it is coerceable or not, otherwise do the following:

set the mode under consideration to be ms, then do

(1) determine the state (or states) to be taken by checking the conditions of

(a) the mode under consideration,

(b) the form of mr and

(c) the coercend.

If this is not the starting step (i.e. if the mode under consideration is not ms), then the state taken must be led to through an arrow from the previous state. (note that the thick bars are of both directions while the other lines are one way only). If no proper state can be found then, mr is not coerceable to ms and halt; otherwise,

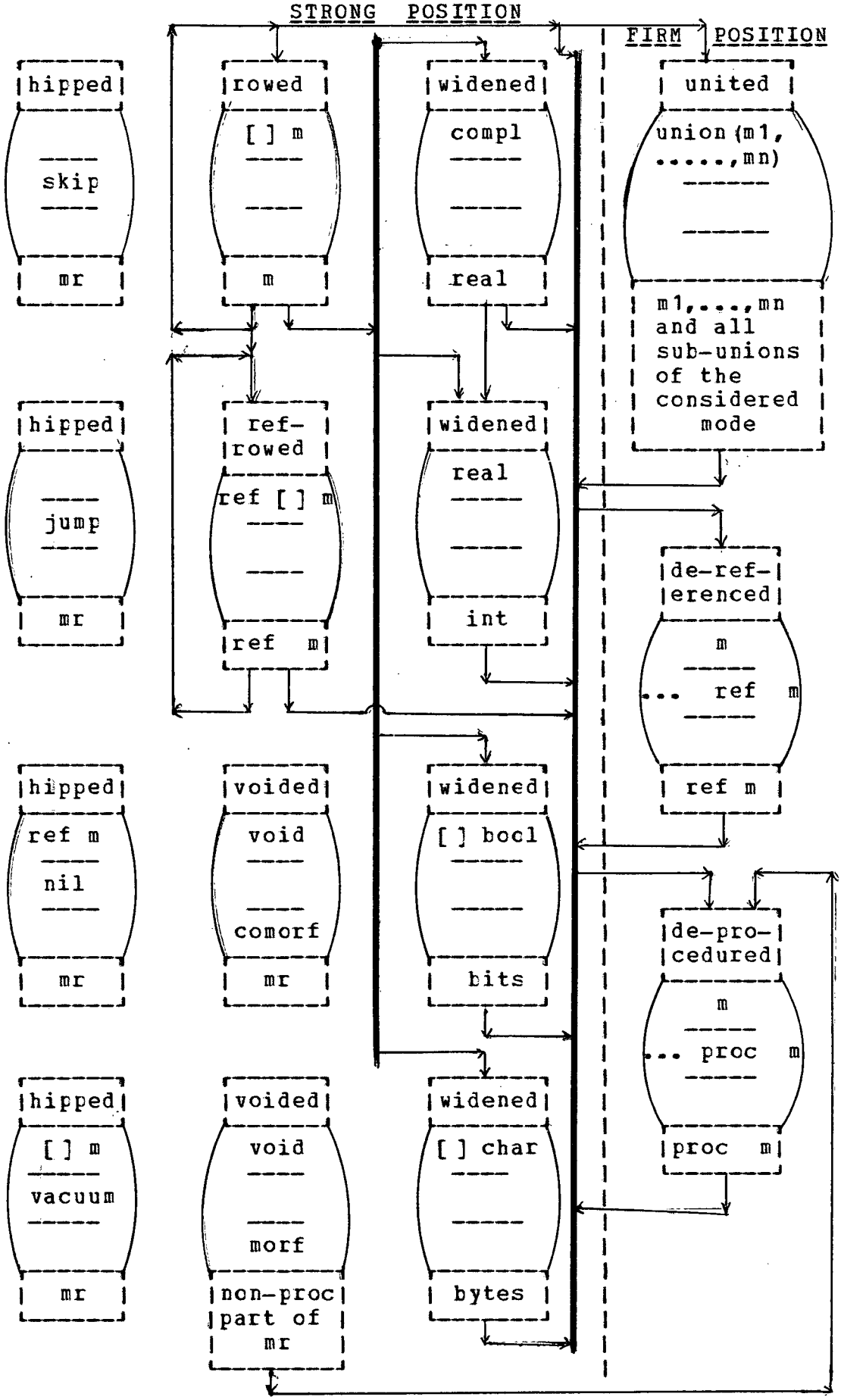
(2) transform the considered mode as specified, and

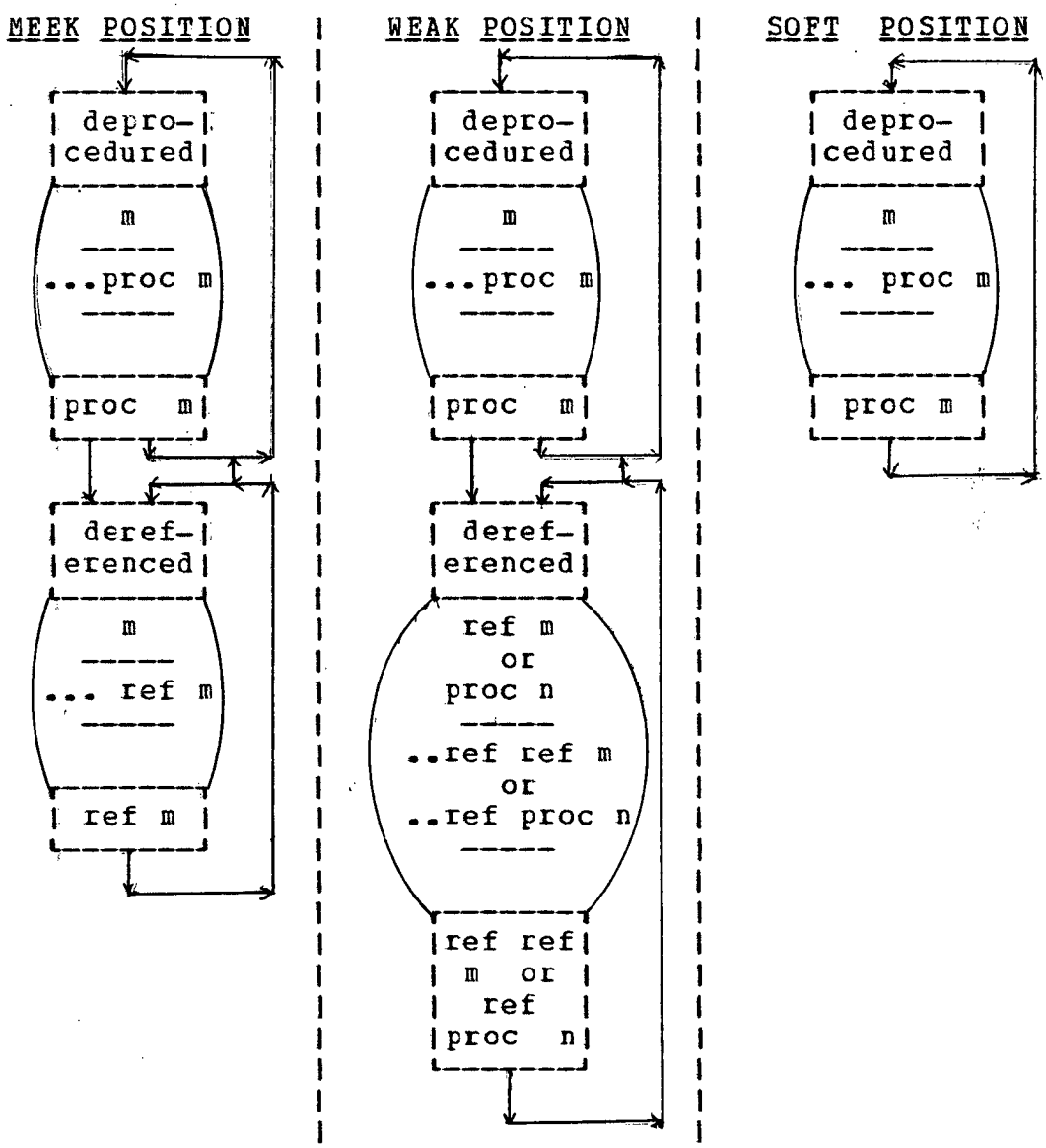
(3) if the considered mode

is the same as mr then, mr is coerceable to ms and halt;

otherwise, goto (1).

The step to be taken may not be unique. For example: if the considered mode is row rowof bool, then both states for 'rowed' and 'widened' have to be considered.


STRONG POSITION
FIRM POSITION
hipped
skip
mr
rowed
[ ] m
m
widened
compl
real
united
union (m1, ..., mn)
m1, ..., mn and all sub-unions of the considered mode
hipped
jump
mr
ref-rowed
ref [ ] m
ref m
widened
real
int
de-ref-erenced
m
... ref m
ref m
hipped
ref m
nil
mr
voided
void
comorf
mr
widened
[ ] bocl
bits
de-pro-cedured
m
... proc m
proc m
hipped
[ ] m
vacuum
mr
voided
void
morf
non-proc part of mr
widened
[ ] char
bytes


MEEK POSITION
depro-
cedured
m
...proc m
proc m
deref-
erenced
m
... ref m
ref m
WEAK POSITION
depro-
cedured
m
...proc m
proc m
deref-
erenced
ref m
or
proc n
..ref ref m
or
..ref proc n
ref ref
m or
ref
proc n
SOFT POSITION
depro-
cedured
m
... proc m
proc m

The algorithm for coercion:

Input: mr: the a priori mode (the source mode).

ms: the a posteriori mode (the target mode) or a null mode list.

sort: the position. This is strong, firm, meek, weak or soft.

coercend: this tells whether it is a comorf or not.

Output: a logical value to tell whether 'mr' is 'sort'ly coerceable to 'ms' or not.

Function: if a coercion step can be taken, the coercion word is stored in the 'cw'-field of the mode. If 'mr' is 'sort'ly coerceable to 'ms' then the coercion sequence is given.

Step1: let 'return' be a logical value.

Combine the grammar rules for 'ms' and 'mr' and apply the equivalence algorithm.

Step2: if 'sort' is " ", then let the first prefix other than ref or proc of 'mr' be pi, if pi is not row or rowof and pi is not procp then goto step19, if pi is row or rowof then 'sort':=weak otherwise 'sort':=firm.

Step3: if 'ms' is null do step3.a to step3.d otherwise goto step5.

Step3.a: let x be 'mr'.

Step3.b: if x is proc m, then search-insert x to 'ms', i.e., if x is not in 'ms' then insert x to the list 'ms' in order, let x = m and goto step3.b;

otherwise,

step3.c: if x is ref m then do step3.c.1 to step3.d otherwise goto step3.d.

Step3.c.1: if 'sort' is firm or meek then, search-insert x to 'ms', let x = m and goto step3.b; otherwise,

step3.c.2: if 'sort' is weak and if m is proc n or ref n then, search-insert x to 'ms' , let x = m and goto step3.b; otherwise,

step3.d: search-insert x to 'ms' and goto step22.

Step4.a: let n = 'mr'.

Step4.b: mark n. If n is ref x or proc x and that n can be 'sort'ly deprocedured or dereferenced, then let n = x and goto step4.b; otherwise, go to step5.

Note that when a mode begins with proc, it can be 'sort'ly deprocedured; when a mode begins with ref, it can be strongly, firmly or meekly dereferenced, or weakly dereferenced when the prefix following ref is proc or ref. The algorithm is similar to step10 to step17.

Step5: if 'sort' is "soft" then goto step12.

If 'sort' is "weak" then goto step14.

If 'sort' is "meek" then goto step16.

Step6: if 'sort' is "strong" and 'mr' is "skip" or "jump", then flag 'mr' and goto step10.

Step7: if 'sort' is "strong" and 'mr' is 'nil' then if 'ms' starts with 'ref' then flag 'mr' and goto step10.

Step8: if 'sort' is "strong" and 'mr' is "vacuum", if 'ms'

starts with 'row' or 'rowof' then flag 'mr' and goto step10.

Step9: flag the nonterminal symbols in the combined grammar as follows:

If 'sort' is "strong" then start in state I otherwise start in state II. Each state gives the flagging instructions and tells what further nonterminals must be considered or what further instructions must be followed. The process stops when there are no more nonterminals to be considered, or when there are some more terminals to be considered and the nonterminal being considered is marked.

STATE I: flag the nonterminal considered. Consider new nonterminals or follow the instructions as follows:

if the right side of the rule is	then consider	in STATE
row m	m	I
rowof m	m	I
rowof bool	bits	III
rowof char	bytes	III
union m(1) ... m(k)	m(1), ..., m(k)	II
ref m	m	IV
compl	real	I
real	int	I
void	goto step18	
other	no operation	

STATE II: flag the nonterminals considered. Consider new nonterminals as follows:

if the right side of the rule is	then consider	in STATE
void	unflag void and goto step10	
union m(1) ... m(k)	m(1), ...,m(k)	II
other	no operation	

STATE III: flag the nonterminal. No new nonterminals to be considered.

STATE IV: do not flag the nonterminal. If the rule for the considered nonterminal does not start with 'row' or 'rowof' then there is nothing to do. If the right side of the rule is row m or rowof m then, if ref m is in the grammar then flag the nonterminal ref m, consider the terminal m in STATE IV (even if ref m is not in the grammar).

step10: let n = 'mr'.

Step11: consider the nonterminal n. If the nonterminal considered is flagged (in step9, step7, step8 or

step6), then 'ms' is STIRMly coerceable from 'mr' and stop. If the rule for the considered nonterminal is ref m or proc m, then set n = m and goto step11; otherwise, stop and 'ms' is not STIRMly coerceable from 'mr'.

Step12: let n = 'mr'.

Step13: if n = 'ms' then goto step22. If n = proc m then set n = m and goto step13, otherwise goto step20.

Step14: let n = 'mr'.

Step15: if n = 'ms' then goto step22. If n = proc m then set n = m and goto step15. If n = ref m and m = ref p or m = proc p then set n = m and goto step15. Goto step20.

Step16: let n = 'mr'.

Step17: if n = 'ms' then goto step22. If n = proc m or n = ref m then set n = m and goto step17, otherwise goto step20.

Step18: (a) if the a posteriori mode is not void then unflag void, the mode under consideration and goto step10 (R.8.2.3.1.a), (after voiding, no coercion may take place) (R.8.2.4.1.and R.8.2.6.1); otherwise

(b) if the coercend 'mr' is a comorf, then flag 'mr', the a priori mode (R.8.2.8.1.a) and goto step10; otherwise,

(c) if the a priori mode 'mr' ends with non-proc void (where non-proc is any prefix other than proc) then unflag 'void' (R.8.2.2.1.a) (only deproceduring

can go before voiding R.8.2.1.1) and go to step10; otherwise,

(d) if the a priori mode 'mr' is not 'void' or does not end with proc void then flag the mode fcllowing the last proc of 'mr' (R8.2.1.b)and goto step10.

Step19: give an error message.

Step20: return := false.

Step21: 'return' is the logical value to be returned. Stop.

Step22: return:=true and goto step21.

End of coercion algorithm.

4.3. An example to show how the coercion procedure works:

In the program, the flag-field of a mode is used for marking the terminal in step4 and planting the flags in step6, step7, step8 and step9 of the algorithm, the cw-field is to store the coercion word and the link-field is used as a backward link so that the coercion sequence can be obtained from the cw-fields of the modes through the link-field.

In the program, the command to check whether mode MR is SORTly coerceable to mode MS is "COERCE" followed by the a priori mode, the a posteriori mode, the position and the coercend which tells whether 'mr' is a COMORF or not. The a priori mode is a mode represented by a number and so is the a posteriori mode. The position is "strong", "firm", "meek", "weak" or "soft". The coercend is either "comorf"

or " ".

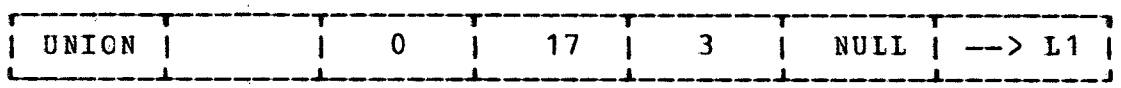
Example: suppose the grammar entered is

```
16   "proc"   17
17   "ref"    2
18   "union"  1 2 3 -2
-2.
```

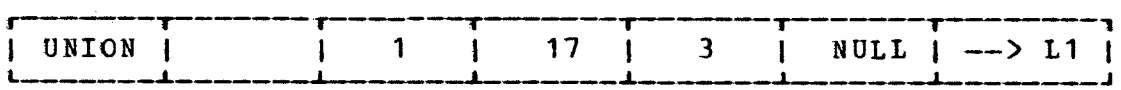
If the command

"COERCE" 16 18 "firm" " "

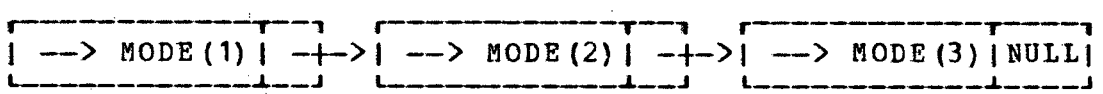
is entered then proc ref int, ref int and int are marked and then MODE(18) is changed from



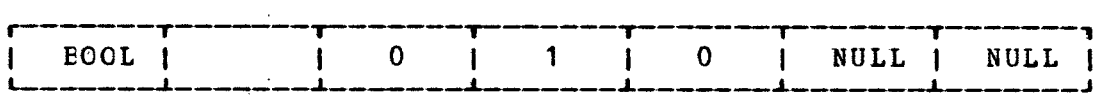
to



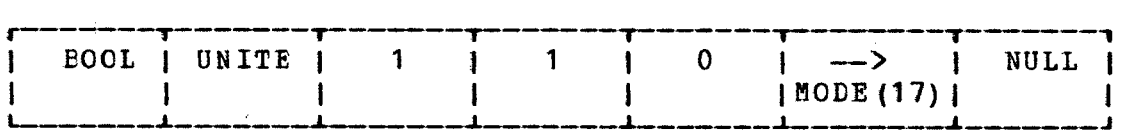
where L1 is



MODE(1) is changed from



to



MODE(2) is changed from

INT		0	2	0	NULL	NULL

to

INT	UNITE	1	2	0	--> MODE(17)	NULL

MODE(3) is not considered though it is in the list of modes to be considered, because MODE(2) is marked. Then MODE(16) is considered: the flag-field is #0, and the terminal is "proc", the coercion word is deprocedure. MODE(17) is considered: the flag-field is #0, and the terminal is "ref", the coercion word is dereference. MODE(2) is considered, and flag-field is #1; so MODE(18) is firmly coerceable from MODE(16) and the coercion sequence is deprocedure, dereference, unite.

If the command

"COERCE" 2 8 "strong" " "

is entered, then int is marked and MODE(8) is changed from

STRUCT		0	8	2	NULL	--> L2

to

STRUCT		1	8	2	NULL	--> L2

where L2 is

--> MODE(10)	-+->	-->MODE(11)	NULL

MODE(3) is changed from

REAL		0	3	0	NULL	NULL

to

REAL	WIDEN	1	3	0	—> MODE(8)	NULL

MODE(2) is changed from

INT		0	2	0	NULL	NULL

to

INT	WIDEN	1	2	0	—> MODE(3)	NULL

MODE(2) is considered again. Since its flag-field is #1, MODE (8) is strongly coerceable from MODE(2) and the coercion sequence is widen, widen.

CHAPTER V

MODE BALANCING

5.1. Balancing and its related operations:

In serial clauses, collateral clauses and conditional clauses, there is a special feature concerning mode that is reflected by the rules:

R.6.1.1g: suite of FEAT CLAUSE trains: FEAT CLAUSE train; FEAT CLAUSE train, completer, suite of strong CLAUSE trains; strong CLAUSE train, completer, suite of FEAT CLAUSE trains.

R.6.2.1d: firm collateral row of MODE clause: firm MODE balance PACK.

R.6.2.1e: firm MODE balance: firm MODE unit, comma symbol, strong MODE unit list; strong MODE unit, comma symbol, firm MODE unit; strong MODE unit, comma symbol, firm MODE balance.

R.6.4.1d: FEAT choice CLAUSE: FEAT then CLAUSE, strong else CLAUSE; strong then CLAUSE, FEAT else CLAUSE.

This feature is that when the a posteriori mode and the positon of a serial, collateral, or conditional clause are given we have to check whether the constituents of the clause are syntactically correct or not. This is called balancing of the modes (the a priori modes of the constituents) to the a posteriori mode which is called the balanced mode.

In balancing, if a list of a priori modes is given, the a posteriori mode and the sort (position) are also given, (the last two are not necessarily given and this will be discussed later) then we can check whether the list of a priori modes can be 'sort'ly balanced to the a posteriori mode by checking: (1) that each of the a priori modes is strongly coerceable to the a posteriori mode and (2) that one of the a priori modes is 'sort'ly coerceable to the a posteriori mode if sort is not strong.

For example:

```
bool a; real x; x:=(a|3|0.3);
```

where we have a conditional clause, (a|3|0.3), whose position is strong, the a posteriori mode is real and the a priori modes are int and real.

Sometimes the a posteriori mode is not known. For example:

```
int y; real x; x:=(x>0|y|x)+3;
```

where the a posteriori mode of the conditional clause (x>0|y|x) is not known. In such a case, a possible list of a posteriori modes can be supplied. Thus we have to extend the meaning of balancing such that when a list of a posteriori modes which may be null or a singleton, the position and a list of a priori modes are given, balancing is to find a mode or some modes or to identify a mode or some modes of the list of a posteriori modes to which the list of a priori modes can be 'sort'ly balanced.

In y:=(x>0|a|b)(i); whether a(i) and b(i) are slices or calls may not yet be known. In such a case, the position and the a posteriori mode are not given and we have to determine whether the position is weak or soft and to provide a possible list of a posteriori modes.

In addition to the above, collateral clauses should also be taken care of here because a constituent may be a collateral clause e.g.

[1:3] real a1, bool p; a1 := (p|(1,2.2,3)|(1,2,3)); .

If the list of a posteriori modes (possibly null) and the position are given when a collateral display (i.e. a list of a priori modes) is given, the job of the procedure 'collateral' is to find or to identify from the a posteriori mode list, a mode or some modes which can be coerced from a row or structural display made up of the elements of the a priori mode list.

5.2. The algorithms:

In the following algorithms, the operations AND and OR between two boolean vectors produces the vector result of applying them between corresponding elements of the vectors.

Balance Algorithm:

Input: parameterlist: a list of nonterminal symbols which represent the a posteriori modes (may be null).

operandlist: a list of modes which are the a priori modes.

sort: strong, firm, meek, weak or soft.

Output: a boolean vector selecting the elements of the 'parameterlist' for which a 'sort' balance of the 'operandlist' can be found.

Function: if 'sort' is null then assign weak or meek to 'sort' if the a priori mode contains a slice or a call, or give an error message if the a priori mode dosen't. If the 'parameterlist' is null, then a list of possible balanced mode is assigned to the 'parameterlist'.

Step1: ravel the 'operandlist', i.e. if the list contains an element which is a set of modes to be balanced then replace this by the set of modes. This process is similar to the raveling of unions. This can be done because in 'sort' balancing all of the a priori modes must be strongly coerceable to the a posteriori mode and only one of the a priori mode must be 'sort'ly coerceable to the a posteriori mode. If one of the a priori modes is a balance pack, this is either 'sort'ly coerceable or strongly coerceable to the a posteriori mode. In the first case, all the other a pricri modes are strongly coerceable to the a posteriori mode and all the constituents of the balance pack are strongly coerceable with one 'sort'ly coerceable to the a posteriori mode. In the second case, one of the other a priori modes is 'sort'ly coerceable to and

all the others together with the constituents of the balance pack are strongly coerceable to the a posteriori mode.

Step2: if 'sort' is not given, then let mi be the first mode other than skip, jump, nil or vacuum in the 'operandlist'; let the first prefix other than ref or proc of mi be pi. If pi is row or rowof then 'sort' := weak, if pi is procp then 'sort':=firm otherwise goto step11.

Step3: if 'parameterlist' is null and if the 'sort' is not strong and not firm then let k be the number of modes in the 'operandlist' and for i:=1 to k do step3.a to step3.d below; otherwise, goto step4.

Step3.a: let x be the nonterminal representing operand(i).

Step3.b: if x is proc m, then search-insert x to the 'parameterlist', let x = m and goto step3.b; otherwise,

step3.c: if x is ref m then do steps 3.c.1, 3.c.2; otherwise, goto step3.d.

Step3.c.1: if 'sort' is firm or meek then search-insert x to the 'parameterlist', let x = m and goto step3.b; otherwise,

step3.c.2: if 'sort' is weak and if m is proc n or m is ref n then search insert x to the 'parameterlist', let x = m and goto step3.b; otherwise,

step3.d: search-insert x to the 'parameterlist'. Operand(i) has been completely processed in forming

the parameterlist.

Step4: let n be the number of elements in the 'parameterlist'. Let check_hip be a vector of 4 boolean values, which are for 'skip','jump', 'nil' and 'vacuum'. Any one of them is in the 'operandlist' if and only if the corresponding value is true.

Step5: let strong_matrix be a boolean matrix of dimension nXm where m is the number of modes defined. Strong_matrix is formed in the following way:

step5.a : for j:=1 to n do the following:

step5.a.1: unflag all the flags of the modes.

Step5.a.2: create a set of flags as directed by step9 of the coercion algorithm, starting from state I with parameter(j) as the nonterminal considered, and if check_hip is not all false, then the set of flags is modified by following step5, step6, or/and step8 of the coercion algorithm.

Step5.a.3: strong_matrix(i,n-j) := flag(mode(i)), for i=1,...,m.

Step5.a.4: parameter(j) has been completely processed.

Step6: if 'sort' is strong then goto step7 otherwise let 'sort_matrix' be a boolean matrix of dimension nXm. 'sort_matrix' is formed in the following way:

step6.a: for j:=1 to n do the following:

step6.a.1: unflag all the flags of the modes.

Step6.a.2: create a set of flags as directed by step9 of

the coercion algorithm. Start in state II if 'sort' is firm else in state III with parameter(j) as the nonterminal under consideration.

Step6.a.3: 'sort_matrix'(i,n-j) := flag(mode(i)), for i=1,...,m.

Step6.a.4: parameter(j) has been completely processed.

Step7: let A, B be boolean vectors of n elements; A be all true and B all false. Let k be the number of elements in the 'operandlist'.

Step8: for i:=1 to k do step8.a to step8.f.

Step8.a: let C, D be boolean vectors of length n.

Step8.b: if operand(i) is a defined mode then let 11 be the nonterminal of operand(i), C := 11-th row of strong-matrix; otherwise, goto step8.c.

Step8.b.1: let operand(i) be x.

Step8.b.2: if x = ref m or proc m then let 12 be the nonterminal of m, D := 12-th row of strong_matrix, C := C OR D, let x := m and goto step8.b.2; otherwise goto step8.d.

Step8.c: if operand(i) is a collateral clause then C := collateral(parameterlist,the list of modes of the collateral clause, strong) otherwise goto step8.f.

Step8.d: A := A and C.

Step8.e: if 'sort' is strong then goto step8.f.

Step8.e.1: if operand(i) is a defined mode then C := the 11-th row of 'sort_matrix' otherwise goto

step8.e.5.

Step8.e.2: let operand(i) be x.

Step8.e.3: if x = proc m or if x = ref m and x can be 'sort'ly dereferenced then

D := 12-th row of 'sort_matrix'; otherwise goto step8.e.6.

Step8.e.4: C := C OR D, let x := m and go to step8.e.3.

Step8.e.5: C := collateral(parameterlist, the list of modes of the collateral clause, 'sort').

Step8.e.6: B := B OR C.

Step8.f: operand(i) has been completely processed.

Step9: if 'sort' is not strong then

A := A AND B.

Step10: return the boolean vector A and stop.

Step11: give an error message. Let A be all false. Goto step10.

End of the balance algorithm.

Collateral Algorithm:

Input: parameterlist: a list of nonterminal symbols of modes which are the a posteriori modes.

operandlist: a list of nonterminal symbols of modes which are the a priori modes.

sort: strong, firm, meek, weak or soft.

Output: a boolean vector selecting the elements of 'parameterlist' which can be 'sort'ly balanced from the collateral mode display 'operandlist'.

Function: if the 'parameterlist' is null, a list of possible balanced mode is assigned to 'parameterlist'.

Step1: create a boolean vector 'row', of which an element is true if and only if the corresponding element of 'parameterlist' begins with row or rowof. If 'sort' is strong, create a boolean vector 'struct' of which an element is true if and only if the corresponding element of 'parameterlist' begins with struct. If 'parameterlist' is null then insert all the defined modes beginning with row or rowof to the 'parameterlist'.

Step2: let 'plist' be a list of modes such that 'plist' and 'parameterlist' are of the same length and if an element of 'parameterlist' is row m or rowof m then the corresponding element of 'plist' is m, otherwise the corresponding element of 'plist' is the pseudo-mode (mode0) which is not a defined mode, but considered as a mode in the list. Let 'qlist' be a copy of 'operandlist'.

Step3: let B be a boolean vector.

B := balance(plist,qlist,sort).

Step4: if 'sort' is not strong or 'struct' is a vector of false values then goto step7. If 'sort' is strong and 'struct' not all false then let 'plist' be a list of modes such that 'plist' and 'parameterlist' are of the same length and if an element of 'parameterlist' is struct m1 m2 ... Mk and k is

equal to the number of entries in 'operandlist' then the corresponding element of 'plist' is struct m1 ... Mk, otherwise the corresponding element of 'plist' is the pseudo mode.

Step5: let A be a boolean vector of which an element is true if and only if the corresponding element of plist is a mode (i.e. not the pseudo mode). If A is all false then goto step6.

For i:=1 to k do step5.a to step5.g.

Step5.a: create a new mode list 'rlist' of the same length as 'plist' such that if an element of 'plist' is a defined mode then the corresponding element of 'rlist' is the i-th field of that mode of 'plist' (which is a structure) otherwise it is the pseudo-mode mode0.

Step5.b: for each mode other than mode0 of 'rlist', jump over the selector to get the mode.

Step5.c: create a boolean vector 'check_hip' as in step4 of the balance algorithm.

Step5.d: let strong_matrix be a boolean matrix of dimension nXm as that in step5 of the balance algorithm by doing step5.a of balance algorithm with n := number of elements in 'rlist' (the same as in 'parameterlist').

Step5.e.1: let c be a boolean vector of length n.

Step5.e.2.a: if operand(i) is a defined mode then, let l1 be the nonterminal of operand(i), C := l1-th row of

strong_matrix; otherwise goto step5.e.3.

Step5.e.2.b: let operand(i) be x.

Step5.e.2.c: if x = ref m or x = proc m then, let 12 be the nonterminal of m, D := 12-th row of strong_matrix, C := C OR D, let x = m and goto step5.e.2.c; otherwise goto step5.f.

Step5.e.3: if operand(i) is a collateral clause then C := collateral(rlist, list of modes of the collateral clause of operand(i), "strong"); otherwise,

step5.e.4: if operand(i) is a serial or a conditional clause then,

C := balance(rlist, list of modes of the clause of operand(i),"strong"); otherwise goto step5.g.

Step5.f: A := A AND C, if A is all false, then goto step7.

Step5.g: operand(i) has been completely processed that is the i-th field of the structure has been completely processed.

Step6: B := B OR A

step7: return the boolean vector B and stop.

End of the collateral algorithm.

5.3. Examples to show how the procedures work:

In the program, instead of a boolean matrix, a vector of bitstrings is formed so that a row of the matrix in the algorithm is but a bitstring. 'balance' and 'collateral' are two subroutines and bitstrings are returned by them.

When the command is "BALANCE" followed by the operandlist, the sort and the parameterlist, 'balance' is called and a sublist of the parameterlist (if it was not null) or a list of modes to each of which the operandlist can be balanced will be given. When the command is "COLLATERAL" followed by the operandlist, the sort and the parameterlist, 'collateral' is called and a sublist of the parameterlist or a list of modes to each of which the operand display can be balanced will be given.

Example: suppose the mode grammer entered is

```
16   "ref"   3
17   "proc"  3
18   "ref"   17
19   "rowof" 3
20   "rowof" 2
-2.
```

When the command

```
"BALANCE" 17 18 19 -2 "firm" -2
```

is entered then

(1) As MODE(17), MODE(18) and MODE(19) are modes, nothing is done by 'ravel'.

(2) The parameterlist is null and sort is firm, then the parameterlist is formed as follows:

MODE(17) is ref real, so ref real and real are inserted to the 'parameterlist'.

MODE(18) is ref proc real, so ref proc real, proc real and real are inserted to the 'parameterlist'.

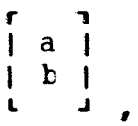
MODE(19) is rowof real, so rowof real is inserted to the parameterlist.

The 'parameterlist' is now real, ref real, proc real, ref proc real, and rowof real.

(3) the strong_matrix is formed as follows:

(i) 'check_hip' is #0 as none of skip, jump, nil and vacuum appears in the operandlist.

(ii) the last column (i.e. the 32nd column) is (00010000000000000000)*t where (ab)*t means



Since real is the a posteriori mode in 'post' only the flag field of real is set to #1, that is flag(MODE(3)) is #1, therefore the fourth element is 1, the others are zero for this column.

(iii) the 31st column is (00000000000000001000)*t since ref real, that is MODE(16) is the a posteriori mode in 'post' so the 17th element of this column is 1, the others are zero for this column.

(iv) similarily the 30th column is (00000000000000000100)*t,

the 29th is (00000000000000000010)*t and the 28th is (00010000000000000001)*t. Entries elsewhere are zero.

(4) the sort_matrix is formed like strong_matrix and the sort_matrix is the same as strong_matrix except that sort_matrix(3,28) is 0 while strong_matrix(3,28)is 1. This is because in posting flags from rowof real, real is flagged

for strong but not for firm.

The strong_matrix is as follows:

```
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000010001|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000010|
|00000000000000000000000000000100|
|00000000000000000000000000001000|
|00000000000000000000000000010000|
```

The firm_matrix is as follows:

```
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000001|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000010|
|00000000000000000000000000000110|
|00000000000000000000000000001110|
|00000000000000000000000000011110|
```

(5) A = (11111111111111111111111111111111),

B = (00000000000000000000000000000000).

(6) operand(1), proc real is considered

C = (00000000000000000000000000000100) OR
(00000000000000000000000000010001)
= (00000000000000000000000000010101)

A = A AND C
= (11111111111111111111111111111111) AND
(00000000000000000000000000010101)
= (00000000000000000000000000010101)

C = (00000000000000000000000000000100) OR
(00000000000000000000000000000001)
= (00000000000000000000000000000101)

B = B OR C
= (00000000000000000000000000000101).

(7) operand(2), ref proc real is considered

C = (00000000000000000000000000001000) OR
(00000000000000000000000000000100) OR
(00000000000000000000000000010001)
= (00000000000000000000000000011101).

A = A AND C
= (00000000000000000000000000010101) AND
(00000000000000000000000000011101)
= (00000000000000000000000000010101)

C = (00000000000000000000000000001000) OR
(00000000000000000000000000000100) OR
(00000000000000000000000000000001)
= (00000000000000000000000000001101).

B = B OR C

= (00000000000000000000000000000101) OR

(00000000000000000000000000001101)

= (00000000000000000000000000001101).

(8) operand(3), rowof real is considered

C = (00000000000000000000000000010000)

A = A AND C

= (00000000000000000000000000010000)

C = (00000000000000000000000000010000)

B = B OR C

= (00000000000000000000000000011101)

(9) A = A AND B

= (00000000000000000000000000010000) AND

(00000000000000000000000000011101)

= (00000000000000000000000000010000).

(10) The list to be returned contains one element: rowof real, i.e. rowof real is the balanced mode.

When the command is

"COLLATERAL" 2 2 3 -2 "firm" 19 20 -2,

then

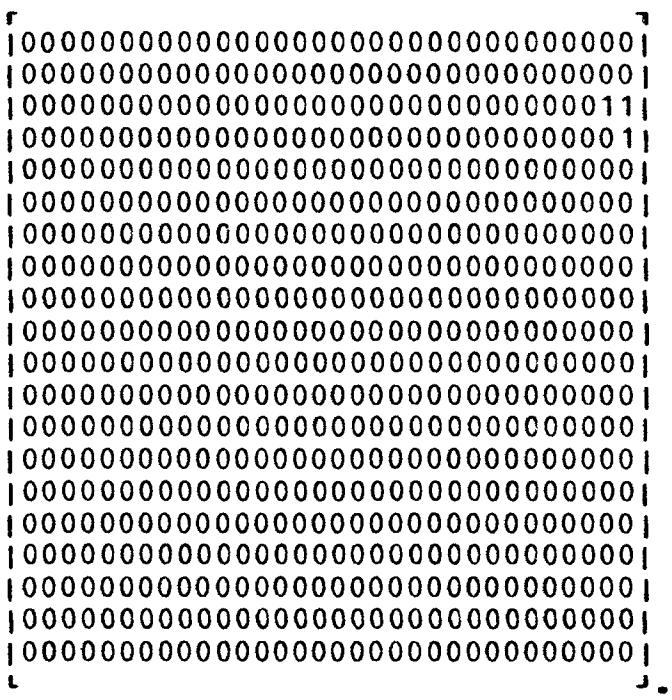
(1) 'row' = (00000000000000000000000000000011),

'struct' = (00000000000000000000000000000000).

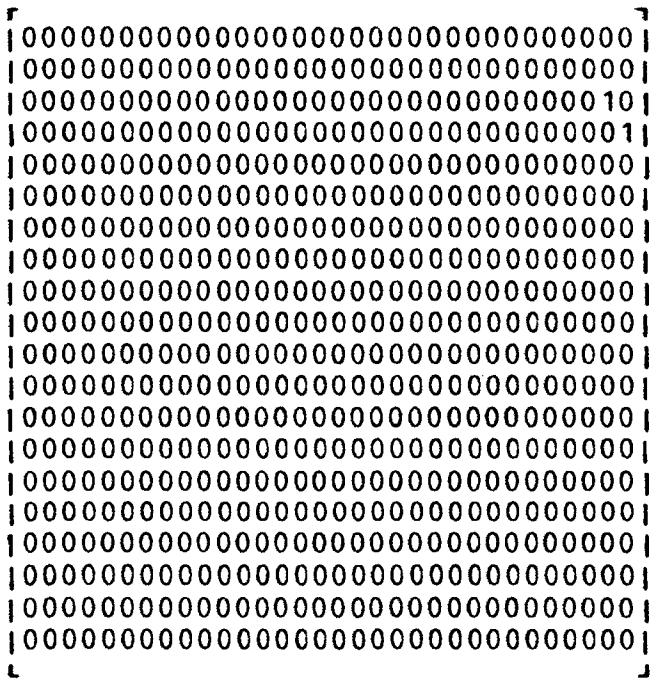
(2) 'plist' = 3 2 -2 i.e. the list contains real, int.

(3) B = balance(plist,operandlist,sort) where the operandlist is int, int, real.

The strong_matrix is:



The sort_matrix is:



So

A = (00000000000000000000000000000011) AND

(00000000000000000000000000000011) AND

(00000000000000000000000000000001)

= (00000000000000000000000000000001)

B = (00000000000000000000000000000010) OR

(00000000000000000000000000000010) OR

(00000000000000000000000000000001)

= (00000000000000000000000000000011)

A = A AND B

= (00000000000000000000000000000001)

(4) The list to be returned contains rowof real only.

CHAPTER VI

OPERATOR IDENTIFICATION

6.1. Operator identification:

ALGOL 68 allows the programmer to define new operators or to extend the definition of existing operators to cover modes which he has defined so that he may make full use of the different defined modes. Because of this, the following problem arises. In an applied occurrence of an operator, this operator may have been defined more than once, for example, the operator + has had at least 16 definitions in the standard prelude, the question is how to chcose the correct definition for the operator. The process of choosing the correct operator in an applied occurrence is called the identification of the operator-defining occurrence of the operator [R.4.3.2b, R.4.4c]. An applied occurrence of an operator occurs in a formula where the modes of the operands (a priori modes) are given and naturally they may be conditional clauses, closed clauses, collateral clauses or serial clauses whose a priori modes are given. The syntax of a formula [R.8.4.1] allows each operand to be a firm MODE tertiary. Thus to identify an operator is to find out the defining occurrence of the operator such that the modes of the parameters in that defining occurrence are firmly coerceable from the corresponding operands of the applied occurrence of the

operator. Also the defining occurrence identified should be unique (a further discussion is in chapter 7).

6.2. The algorithm:

In this algorithm, the correct operator symbol and the correct number of parameters are assumed.

Input: left operand: the mode of the left operand of the operator in the applied occurrence. This may be a conditional/serial clause or collateral clause. It is null if the operator is monadic.

right operand: the mode of the right operand of the operator in the applied occurence. This is similar to the left operand except that it cannot be null.

eligibleoplist: a list of operators denoted by mode pairs, each of which is for the left and the right parameter of a defining occurence of the operator.

Output: a boolean vector which selects from the eligibleoplist those identified by the operands.

Step1: make a list, 'parameterlist' of nonterminals of modes of the right parameters of the 'eligibleoplist'.

Step2: let A,B be boolean vectors of n elements, where n is the number of entries in the parameterlist. If 'right operand' requires balancing, then goto step12 otherwise

step3: let firm-matrix be an mXn boolean matrix where m is the number of defined modes, n is the number of entries in the parameterlist; firm-matrix is formed in the following way:

for j=1 to n do step3.a to step3.d.

Step3.a: unflag all the flags of the modes.

Step3.b: create a set of flags as directed by step9 of the coercion algorithm starting from STATE II with the j-th entry of the parameterlist as the mode considered.

Step3.c: firm-matrix(i,n-j) := flag(mode(i)) for i = 1,...,m.

Step3.d: the j-th entry of the parameterlist has been completely processed.

Step4: let l be the nonterminal of the right operand, A be a boolean vector of n false values.

Step5: A = A OR the l-th row of firm-matrix. If l is ref m or proc m then let l be the nonterminal of m and goto step5; otherwise,

step6: if the operator is monadic then goto step11; otherwise, replace the parameterlist by a list of nonterminals representing the left parameters of the 'eligibleoplist'.

Step7: if the left operand requires balancing then go to step14; otherwise,

step8: the same as step3 (including step3.a to step3.d).

Step9: let l be the nonterminals of the 'left operand', B a boolean vector of n false values.

Step10: B = B OR the l-th row of firm-matrix, if l is ref m or proc m then let l be the nonterminal of m and goto step10, otherwise A = A AND B.

Step11: A is the vector to be returned. Stop.

Step12: make a list, 'olist' of the operands inside the clause of the right operand.

Step13: if the right operand is a serial or conditional clause then A = balance(parameterlist, olist, "firm") otherwise A = collateral(parameterlist, olist, "firm"). Goto step6.

Step14: make a list 'olist' of the operands inside the clause of the left operand.

Step15: if the left operand is a serial or conditional clause then B = balance(parameterlist,olist,"firm") otherwise B = collateral(parameterlist,olist,"firm"). A = A AND B. Goto step11.

End of the algorithm of operator identification.

6.3. An example:

In the program, a bitstring is used for a boolean vector as in balance and collateral. The list of mode pairs is terminated by -2 -2 or any two numbers less than -2. If the operator is monadic, then the list is

-2 n1 -2 n2 -2 n3 ... -2 -2

where n1,n2... Are nonterminals for the modes of the right parameters of the operator, -2 may be replaced by any number less than -2. When the command is "OPERATORS" followed by the left operand, the right operand and the pairs of parameters, a sublist of 'operators' which can be identified by the operands will be given.

For example:

suppose the mode grammar entered is

```
16 "ref" 3
17 "proc" 3
18 "ref" 17
-2
```

and the command

```
"OPERATORS" -2 3 -2 3 -2 16 -2 -2
```

is entered,

(1) The firm_matrix is

```
┌                                                  ┐
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000001|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
|00000000000000000000000000000010|
|00000000000000000000000000000000|
|00000000000000000000000000000000|
└                                                  ┘
```

(2) A = (00000000000000000000000000000000)

(3) A = (00000000000000000000000000000000) OR

(00000000000000000000000000000001)

= (00000000000000000000000000000001).

(4) The operator is monadic. The operator identified is +real.

CHAPTER VII

CONCLUDING REMARKS

This is a model in ALGOL W that shows how modes can be manipulated in an ALGOL 68 implementation. This is done according to the revised syntax rules. The main changes in the revised syntax are: void is a symbol (still not a mode), no proceduring in coercion, the coercion to vacuum is hipping, the definition of vacuum and nonproc are different, hipping is not an explicit coercion, however implicitly it is, and a compiler should take care of it.

The definition of equivalent modes is not given in [R] and not included in the new syntax yet. The definition given in chapter 3 is one modified, from that given by Mary Zosel in p.7 of [Z], which is not exact as we can see the modes u and v defined below are not equivalent by the definition.

Mode u = union (bool, procp (u) int) and
mode v = union (bool, procp (u) int, procp (v) int).
But they are equivalent by her algorithm. The definition is not applicable owing to (1) the representation of a union mode is not unique, (2) a union with two constituents may be equivalent to a union with more than two constituents. The present definition is only a suggested one, probably there will be a proper definition in the Revised Report.

There are restrictions (R.4.4.2c and R.4.4.3c) in

defining occurrences of operators, and they have been shown not powerful enough to guarantee the uniqueness in identification of operators in its applied occurrence [W], [L1]. In identification of operators, one and only one operator should be identified. In this work, a list of operators identified is delivered, if this is null then no defining occurrence can be identified, if this contains more than one entry, then the operator is ambiguous.

In this model, the number of balanced modes or the number of defining occurrences of an operator is at most 31. When this is applied in a real compiler, in balancing, if the balanced mode is given then the number of balanced modes is one, if not, then a list of possible balanced modes that contains the a priori modes and modes 'sort'ly deferenced or 'sort'ly deprocedured from them will be supplied. Thus 31 is a number big enough for that. In operator identification, if the number of defined occurrences of the operator is greater than 31, then the list of operators to be identified can be broken up into sublists of operators of less than 31 elements so this restriction can be handled.

If there is no 'firm collateral display' (which may be included in the revised Report), the subroutine 'collateral' will be much simpler and the subroutine 'pick' can be simplified too.

REFERENCES

[A] Algol W Programming Manual, University of Newcastle upon Tyne, 1970.

[K] Koster, C. H. A. On Infinite Modes, Algol Bulletin AB 30.3.3, Feb. 1969 pp. 86-89.

[L] Lindsey, C. H. and van der Meulen, S. G. Informal Introduction to ALGOL 68, North-Holland Publishing Company, Amsterdam, 1971.

[L1] Lindsey, C. H. Displays and loosely related, IFIP WG 2.1, No. 3 (168), Novosibirsk, Sept. 1971.

[P] Peck, J. E. L. Some steps in compiling ALGOL 68, Gesellschaft fur Informatik, Bericht Nr. 4, Saarbrucken, 7-9 March, 1972.

[P1] Peck, J. E. L. An ALGOL 68 Companion, University of British Columbia, 1971.

[P2] Peck, J. E. L. On storage of modes and some context conditions, Proc. Informal Conference on ALGOL 68 Implementation, University of British Columbia, Aug. 1969 pp. 70-77.

[R] van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., and Koster, C. H. A. Report on the algorithmic language ALGOL 68, Mathematisch Centrum, MR101, Amsterdam, Oct., 1969, and Numerische Mathematic, 14, 1969 pp. 79-218.

[S] Scheidig, H. Coercions in ALGOL 68, Technische Hochschule, Munchen, Feb. 1970.

[W] Wossner,H. On identification of operators in ALGOL 68, ALGOL 68 Implementation, Peck (Ed.), North-Holland Publishing Company, Amsterdam, 1971, pp.111-118.

[Z] Zosel, M. A formal grammar for the representation of modes and its application to ALGOL 68. University of Washington, Oct. 1971.

APPENDIX A

THE PROGRAM

```
BEGIN COMMENT date ---------- 1973 April ----------;
RECORD md(STRING(6)t,cw; BITS flag;
          INTEGER mn, nf; REFERENCE(md,nd)link,field);
COMMENT --- 'md's are used for the internal storage of
modes. The field 't' is the terminal, 'mn' is the mode
number, 'nf' is the number of fields and 'field' is a
reference to another 'md' (the field) or a reference to
an 'nd' list (the fields). The parts 'cw', 'link' and
'flag' are dynamic, i.e. are used for temporary storage
during the mode manipulation algorithms. 'cw' is used
to store coercion steps, 'link' to link modes together
in lists and 'flag' for various flags;
RECORD nd(REFERENCE(md)v; REFERENCE(nd)nxtmd);
COMMENT --- 'nd' is used for simple lists of modes;
RECORD ndo(STRING(6)op_symbol; REFERENCE(md)l_par,r_par;
           REFERENCE(ndo)nxtop);
COMMENT --- the nodes 'ndo' are used for lists of operators;

COMMENT --- the procedures are related according to the
following grammar:

                        <<<utility>>>
      clean_up :.
      clean_flag : .
      searchin : searchin.
      mpr : .
      ravel : select, ravel.
        select : select.
      post : consider, hip_link, unions, search.
        consider : .
        hip_link : .
        unions : consider.
        search : equal.
          equal : equal.
      possible : searchin, fit.
      fit : .

                      <<<equivalence>>>
      equivalence : context, ravel, join_class, pass,
                    searchin, related, clean_flag.
        context : context.
        join_class : .
        pass : scan.
          scan : equivu, equivl.
            equivu : elmu.
              elmu : elmu.
            equivl : equivl.
        related : searchin.
```

```
                    <<<coercion>>>
coerce : possible, fit, post.

          <<<operator  identification>>>
op_ident : operand_ident, clean_flag.
  operand_ident : make_pars, form_matrix,
                  check_hip, which.
    make_pars : make_pars.
    form_matrix : post.
    check_hip : check_hip.
    which : id, clean_up, collateral, bala.
      id : fit .
      collateral : searchin, fieldlister, pick,
                   balance, check_hip, form_matrix,
                   which, clean_flag.
        fieldlister : fieldlister.
        pick : pick.
        balance : ravel, possible, form_matrix,
                  check_hip, clean_flag, bala.
          bala : which.

                    <<<testing>>>
test_them : equivalence, reverse_ravel, related,
            clean_flag, coerce, clean_up, select,
            identify, balance, collateral, selecto,
            op_ident, enter_grammar.
  identify : form_matrix, id.
  reverse_ravel : include.
    include : include.
  selecto : selecto.
  enter_grammar : mpr.
```

Each of the above procedures may call some input or output procedures which are not stated, in addition to read, readon, write and writeon. The I/O procedures are related as follows:

```
                    <<<input>>>
readmdlist : readmd, readmdlist.
  readmd : .
readoplist : readmd.

                    <<<output>>>
```

procedures to save output to the output buffer 'out_line' :

```
saveoplist : saveopnd, saveon.
  saveopnd : save, savemdlist, saveon, save_md.
    savemdlist : saveopnd, saveon.
      saveon : save.
        save : .
    save_md : save, saveon, save_md.
```

The procedure to print out the output buffer 'out_line' :

```
saveout : .      ;
```

```
COMMENT --- 'clean_up' cleans up the fields 'link', 'flag'
and 'cw' in each mode of the grammar;
PROCEDURE clean_up;
  FOR i:=0 UNTIL max DO
    BEGIN link(mode(i)):=NULL; flag(mode(i)):=#0;
    cw(mode(i)):=" " END;

COMMENT --- 'clean_flag' celans up the 'flag' field in each
mode of the grammar;
PROCEDURE clean_flag;
  FOR i:=0 UNTIL max DO flag(mode(i)):=#0;

COMMENT --- 'searchin' searches the 'nd' list 'r' for the
'md' 'p' and inserts it, if necessary, in the order of mode
numbers ('mn');
PROCEDURE searchin(REFERENCE(md) VALUE p;
                   REFERENCE(nd) VALUE RESULT r);
  IF r=NULL THEN r:=nd(p,NULL)ELSE
  IF p=v(r) THEN ELSE
  IF mn(p)<mn(v(r)) THEN r:=nd(p,r) ELSE
  searchin(p,nxtmd(r));

COMMENT --- 'mpr' delivers the string "mi" where "i" is the
value of 'i'. If i<min then it delivers the primitive mode
instead;
string(5) procedure mpr(integer value i);
  BEGIN STRING(15)s; s:=" ";
  IF i<0 THEN ELSE IF i<min THEN
    s(0|6):=t(mode(i)) ELSE
    BEGIN s:=intbase10(i); s(9|1):="m"; s(0|6):=s(9|6) END;
  s(0|5) END mpr;

COMMENT --- 'ravel' ravels the unions ('i'=1) or the
conditional clauses ('i'=2) in the mode list 'p'. In
raveling the unions, it assumes that context con-
ditions have already been checked;
PROCEDURE ravel(REFERENCE(nd) VALUE RESULT p;
                INTEGER VALUE i);
  IF p¬=NULL THEN
    BEGIN IF trace_it THEN
      BEGIN save("%ravel"); savemdlist(p); saveout END;
    IF CASE i OF (t(v(p))="union",
    (cw(v(p))="cond") OR (cw(v(p))="seri") ) THEN
      BEGIN REFERENCE(nd) q,r; q:=r:=CASE i OF
        (select(¬#0,field(v(p))), link(v(p)));
      WHILE nxtmd(r)¬=NULL DO r:=nxtmd(r);
      nxtmd(r):=nxtmd(p); p:=q; ravel(p,i) END
    ELSE ravel(nxtmd(p), i) END;

COMMENT --- 'select' performs the apl operation a/list;
REFERENCE(nd)PROCEDURE select(BITS VALUE a;
                              REFERENCE(nd) VALUE list);
  IF list=NULL THEN NULL ELSE IF a AND #1 = #1 THEN
  nd(v(list), select(a SHR 1, nxtmd(list))) ELSE
```

```
  select(a SHR 1, nxtmd(list));

COMMENT --- 'post' flags 'mr' if 'mr' is "skip" or "jump",
if 'mr' is "nil" and 'ms' starts with "ref", or if 'mr'
is "vacuum" and 'ms' starts with "row" or "rowof" otherwise
traces the a posteriori route through the grammar, from the
mode 'ms', planting the flag 'flag1' as it goes;
PROCEDURE post(REFERENCE(md) VALUE mr,ms; BITS VALUE flag1;
               STRING(6) VALUE sort, coercend);
  BEGIN REFERENCE(nde)cnsd; REFERENCE(md)m, lst;
  INTEGER st; RECORD nde(REFERENCE(nde)nxtnde;
                         REFERENCE(md)mde, last;
                         INTEGER state);
  COMMENT --- 'nde's are used by the queueing mechanism;
  REFERENCE(nd)p;

  COMMENT --- 'consider' queues the mode 'm1' along with
  the state 'st' and saves coercion step 'cwd' in the 'cw'
  field of 'm1';
  PROCEDURE consider(REFERENCE(md) VALUE m1;
                     INTEGER VALUE state;
                     STRING(6) VALUE cwd);
    BEGIN IF cnsd=NULL THEN cnsd:=nde(NULL,m1,m,state) ELSE
      BEGIN REFERENCE(nde) q; q:=cnsd;
      WHILE nxtnde(q)¬=NULL DO q:=nxtnde(q);
      nxtnde(q):=nde(NULL,m1,m,state) END;
    cw(m1):=cwd END consider;

  COMMENT --- 'hip_link' flags 'mr' and links 'mr', 'ms'
  when 'mr' is coerceable to 'ms' by hipping;
  PROCEDURE hip_link;
    BEGIN cw(mr):="hipped"; flag(mr):=flag(mr) OR flag1;
    link(mr):=ms; link(ms):=mode0 END hip_link;

  COMMENT --- 'unions' takes care of the union mode 'm' in
  strong or firm coercion;
    PROCEDURE unions(REFERENCE(md) VALUE m);
    BEGIN REFERENCE(nd)p; p:=field(m); WHILE p¬=NULL DO
      BEGIN consider(v(p),3,"unite"); p:=nxtmd(p) END
  END unions;

  COMMENT --- 'search' searches through the mode grammar for
  the mode equal to 'q'. Only the 't', 'nf' and 'field'
  fields are examined;
  REFERENCE(md) PROCEDURE search(REFERENCE(md) VALUE q);
    BEGIN

    COMMENT --- 'equal' determines whether the 'nd' lists
    'p' and 'q' are equal;
    LOGICAL PROCEDURE equal(REFERENCE(nd)VALUE p,q);
      IF p=q THEN TRUE ELSE IF p=NULL THEN q=NULL ELSE
      (v(p)=v(q)) AND equal(nxtmd(p),nxtmd(q));

    REFERENCE(md)mi; INTEGER i; i:=min;
```

```
  WHILE i<=max DO
    BEGIN mi:=mode(i);
      IF (nf(q)=nf(mi)) AND (t(q)=t(mi)) AND
        (IF field(q) IS nd THEN equal(field(q),field(mi))
        ELSE field(q)=field(mi)) THEN i:=max+2  ELSE
      i:=i+1 END;
  IF i¬=max+2 THEN mi:=NULL;
  mi END search;

cnsd:=NULL; m:=mode0;
consider(ms,
  IF sort="strong" THEN 1 ELSE IF sort="firm" THEN 2
  ELSE 3, " ");
WHILE cnsd¬=NULL DO
  BEGIN st:=state(cnsd); m:=mde(cnsd); lst:=last(cnsd);
  IF trace_it THEN write("*** considering ",
    IF m=NULL THEN "null" ELSE mpr(mn(m))," in state ",
    st, " last was ", IF lst=NULL THEN "null" ELSE
    mpr(mn(lst)));
  cnsd:=nxtnde(cnsd);
  IF(flag(m)=¬#1) AND (t(m)="void") AND
    (sort¬="strong") THEN
  COMMENT --- if void appears as the a posteriori mode in
  a position other than strong and that seems to be
  coerceable, e.g. the a priori mode is ref proc void,
  then a message will be given and the result is not
  coerceable;
  write("invalid a posteriori mode") ELSE
    BEGIN
    CASE st OF
      BEGIN
        BEGIN COMMENT ********** state 1 (strong);
        link(m):=lst; flag(m):=flag(m) OR flag1;
        IF (t(mr)="skip") OR (t(mr)="jump") THEN
          hip_link ELSE
        IF (t(mr)="nil") AND (t(ms)="ref") THEN
          hip_link ELSE
        IF (t(mr)="vacuum") AND (t(ms)="ref") THEN
          hip_link ELSE
        IF t(m)="union" THEN unions(m) ELSE
        IF t(m)="rowof" THEN
          BEGIN consider(field(m),1,"row");
          IF t(field(m))="bool" THEN
            consider(mode(bits1),3,"widen") ELSE
          IF t(field(m))="char" THEN
            consider(mode(bytes),3,"widen") END ELSE
        IF t(m)="row" THEN
            consider(field(m),1,t(m)) ELSE
        IF m=mode(compl) THEN
          consider(mode(real1),1,"widen") ELSE
        IF m=mode(3) THEN
          consider(mode(2),3,"widen") ELSE
        IF t(m)="ref" THEN consider(field(m),4,"refrow")
          ELSE
```

```
            IF t(m)="void" THEN
              BEGIN IF link(m)¬=mode0 THEN
              COMMENT --- no coercion goes after voiding,
              so if the a posteriori mode is void, no
              prefix should go before void;
                flag(m):=flag(m) AND ¬flag1 ELSE
                BEGIN REFERENCE(md) mm, mp; mm:=mr;
                IF (coercend="comorf") OR (coercend="loop")
                  THEN
                  BEGIN flag(mm):=flag(mm) OR flag1;
                  cw(mm):="voided"; link(mm):=m END ELSE
                BEGIN mp:=mr; WHILE field(mp) IS md DO
                  BEGIN mm:=mp; mp:=field(mp) END;
                IF (t(mp)="void") AND (t(mm)¬="proc") AND
                  (mm¬=mp) THEN
                  flag(m):=flag(m) AND ¬flag1 ELSE
                  BEGIN mp:=mr; WHILE field(mp) IS md DO
                    BEGIN IF t(mp)="proc" THEN mm:=field(mp)
                    ELSE IF t(mp)(0|3)="row" THEN
                    mp:=md(" "," ",#0,0,0,NULL,mode0);
                    mp:=field(mp) END;
                  cw(mm):="voided"; link(mm):=
                  IF t(mm)="void" THEN NULL ELSE m;
                  flag(mm):= flag(mm) OR flag1 END END END
            END END state_1;
            BEGIN COMMENT ********** state 2 (firm);
            IF t(m)¬="void" THEN
            COMMENT --- void is assumed not to appear as the
            a posteriori mode in any position other than
            strong;
              BEGIN link(m):=lst; flag(m):=flag(m) OR flag1
              END;
            IF t(m)="union" THEN unions(m) END state_2;
            BEGIN COMMENT ********** state 3;
            IF t(m)¬="void" THEN
              BEGIN link(m):=lst; flag(m):=flag(m) OR flag1
              END END state_3;
            BEGIN COMMENT ********** state 4;
            IF t(m)(0|3)="row" THEN
              BEGIN REFERENCE(md)mq1;
              link(m):=lst; consider(field(m),4,"refrow");
              mq1:=search(md("ref"," ",#0,0,1,NULL,
                             field(m)));
              IF mq1¬=NULL THEN
                BEGIN link(mq1):=lst; m:=mq1; cw(m):="refrow";
                flag(m):=flag(m) OR flag1 END END
            END state_4 END END;
      IF flag(m)=¬#0 THEN cnsd:=NULL END;
    IF trace_it THEN writeon("flag = ",flag(m)) END post;

COMMENT --- 'possible' delivers a list of all modes which
can be 'sort'ly coerced from at least one of the modes in
the list 'p'. Note that it is not called when 'sort' is
"strong" and when 'sort' is "firm" it does not consider
```

```
uniting;
reference(nd) procedure possible(reference(nd) value p;
                                 STRING(6) VALUE sort);
  BEGIN REFERENCE(nd) return, q; return:=NULL;
  WHILE p¬=NULL DO
    BEGIN REFERENCE(md) m; q:=NULL; m:=v(p);
    COMMENT --- 'skip', 'jump', 'nil' and 'vacuum' cannot
    be an a posteriori mode;
    IF ¬((t(m)="skip") OR (t(m)="jump") OR (t(m)="nil") OR
         (t(m)="vacuum")) THEN
      BEGIN searchin(m,q); WHILE fit(m,sort) DO
        BEGIN m:=field(m); searchin(m,q) END;
      WHILE q¬=NULL DO
        BEGIN searchin(v(q),return); q:=nxtmd(q) END END;
    p:=nxtmd(p) END;
  IF trace_it THEN
    BEGIN save("%possible"); savemdlist(p);
    save(sort); saveout END;
  return END possible;

COMMENT --- 'fit' determines whether the coercion step,
dereference or deprocedure, can be taken from the mode 'm';
LOGICAL PROCEDURE fit(REFERENCE(md) VALUE m; STRING(6) VALUE
                      sort);
  IF t(m)="proc" THEN TRUE ELSE
  IF (sort¬="soft") AND (sort¬="weak") THEN t(m)="ref" ELSE
  IF sort="weak" THEN
    BEGIN IF t(m)="ref" THEN
    (t(field(m))="ref") OR (t(field(m))="proc")
    ELSE FALSE END
  ELSE FALSE;

COMMENT --- 'equivalence' contains the mode equivalencing
procedures. During equivalencing the modes are in an 'nd'
list whose first element is 'initial' and whose last element
is 'final': at the beginning, all the modes of the same
terminals are linked together through their 'link' fields
and attached to one of the elements of the 'nd'. At the end
of equivalencing, there are as many elements in the 'nd'
list as there are equivalence classes. A set of equivalent
modes then hangs from each element of the 'nd'. The 'flag'
field of an 'md' is used to record the equivalence class
number;
PROCEDURE equivalence;
  BEGIN

  COMMENT --- 'context' checks for two context conditions.
  It determines whether a mode shows itself and whether
  there are multiple occurrences of the same field selector
  in a structure;
  PROCEDURE context(REFERENCE(md) VALUE m;
                    LOGICAL VALUE ref, struct);
    BEGIN IF flag(m)=#1 THEN
      BEGIN
```

```
    write("context condition error involving the mode");
    save_md(m); saveon(","); saveout;
    write(" which is replaced by 'bool'.");
    COMMENT --- this dangerous situation must be corrected
    so we change the mode to something simple__ "bool";
    field(m):=NULL; t(m):="bool"; nf(m):=0 END ELSE
    BEGIN flag(m):=#1;
    COMMENT --- check for shielding;
    IF t(m)¬="procp" THEN
      BEGIN IF t(m)="struct" THEN
        BEGIN REFERENCE(nd) p,q;
        COMMENT --- check for repeated selectors;
        p:=field(m); WHILE p¬=NULL DO
          BEGIN q:=nxtmd(p); WHILE q¬=NULL DO
            BEGIN IF t(v(p))=t(v(q)) THEN
              BEGIN IF max<35 THEN
                BEGIN v(q):=md("%"," ",flag(v(p)),
                  BEGIN max:=max+1; max END, 1,
                link(v(q)), field(v(q)));
                mode(max):=v(q) END;
              write("context condition error",
              " in the structure"); save_md(m);
              save("multiple"); save("selector");
              save(t(v(p))); saveout END;
            q:=nxtmd(q) END;
          p:=nxtmd(p) END;
        struct:=TRUE END ELSE
      IF t(m)="ref" THEN ref:=TRUE;
      IF ¬(ref AND struct) THEN
        BEGIN IF field(m) IS md THEN
        context(field(m),ref,struct) ELSE
          BEGIN REFERENCE(nd) p; p:=field(m);
          WHILE p¬=NULL DO
            BEGIN context(v(p),ref,struct);
            p:=nxtmd(p) END END END END;
      flag(m):=#0 END END context;

COMMENT --- 'join_class' adds the mode 'mi' to the class
which 'v(q)' belongs. This is used when 'mi' and 'v(q)'
are of the same terminal;
PROCEDURE join_class(REFERENCE(md) VALUE mi;
                     REFERENCE(nd) VALUE q);
  BEGIN REFERENCE(md) x; x:=v(q);
  WHILE link(x)¬=NULL DO x:=link(x);
  link(x):=mi; flag(mi):=flag(x) END;

COMMENT --- 'pass' makes one pass through the 'nd' list.
The parameter 'i' determines which of the two different
kinds of tests for equivalence should be made.
          1  -  test the number of fields,
          2  -  test the class numbers;
PROCEDURE pass (INTEGER VALUE i);
  BEGIN
```

```
COMMENT --- 'scan' makes one scan through an 'md' list.
It considers the initial mode and determines whether any
of the remaining modes of the list is in the same
equivalence class as the initial element. If one is not
then it is transferred to the list hanging from 'final'
and is given a new class number;
PROCEDURE scan (REFERENCE(md) VALUE p; INTEGER VALUE i);
  BEGIN

  COMMENT --- 'equivu' determines whether the united
  modes 'a' and 'b' should be in the same equivalence
  class;
  LOGICAL PROCEDURE equivu (REFERENCE(nd) VALUE a,b);
    BEGIN

    COMMENT --- 'elmu' determines whether the mode 'a'
    is equivalent to a member of the mode list 'b';
    LOGICAL PROCEDURE elmu (REFERENCE(md) VALUE a;
                            REFERENCE(nd) VALUE b);
      IF b=NULL THEN FALSE ELSE
      IF flag(a)=flag(v(b)) THEN TRUE ELSE
      elmu(a,nxtmd(b));

    LOGICAL return; REFERENCE(nd) p; return:=TRUE;
    p:=a; WHILE p¬=NULL DO
      IF elmu(v(p),b) THEN p:=nxtmd(p) ELSE
      BEGIN p:=NULL; return:=FALSE END;
    IF return=TRUE THEN
      BEGIN p:=b; WHILE p¬=NULL DO
        IF elmu(v(p),a) THEN p:=nxtmd(p) ELSE
        BEGIN p:=NULL; return:=FALSE END END;
    return END equivu;

  COMMENT --- 'equivl' determines whether two 'nd' lists
  of modes are such that the corresponding elements are
  in the same equivalence classes. It is called by
  'scan' to deal with the modes whose terminals are
  "struct" and "procp";
  LOGICAL PROCEDURE equivl(REFERENCE(nd) VALUE a,b);
    IF a=NULL THEN b=NULL ELSE
    (flag(v(a))=flag(v(b))) AND
              equivl(nxtmd(a),nxtmd(b));

    REFERENCE(md) q, r; r:=p; q:=link(r);
    WHILE q¬=NULL DO IF CASE i OF (
      IF t(p)="union" THEN TRUE ELSE nf(p)=nf(q),
      IF field(p)=field(q) THEN TRUE ELSE
      IF field(p) IS md THEN
          flag(field(p))=flag(field(q)) ELSE
      IF t(p)="union" THEN equivu(field(p),field(q))
      ELSE equivl(field(p),field(q))
                                          )THEN
      BEGIN r:=link(r); q:=link(r) END ELSE
      BEGIN link(r):=link(q); link(q):=NULL;
```

```
        flag(q):=bitstring(class);
        IF tail IS nd THEN v(tail):=q ELSE link (tail):=q;
        tail:=q; q:=link(r); IF trace_it THEN
          BEGIN save("%tail = "); save_md(tail); saveout
        END END END scan;

  p:=pre; svcl:=class;
  WHILE v(p)¬=NULL DO
    BEGIN IF v(final)¬=NULL THEN
      BEGIN tail:=final:=nxtmd(final):=nd(NULL,NULL);
      class:=class+1 END;
    scan(v(p),i); p:=nxtmd(p) END
   END pass;

REFERENCE(nd) initial, final, p, pre;
INTEGER class, svcl; REFERENCE(nd,md) tail;
COMMENT --- check context conditions and ravel all the
unions;
FOR i:=min UNTIL max DO
  BEGIN context(mode(i),FALSE,FALSE);
  IF t(mode(i))="union" THEN ravel(field(mode(i)),1) END;

COMMENT --- modes are classified by their terminals, i.e.
each element of the 'nd' initial hangs a list of modes,
linked through their link-fields, with the same terminal.
The first 12 classes (from 0-th to 11-th) are for
primitives, the 12-th and the 13-th are for 're' and 'im'
of 'complex'. the 14-th is for 'rowof' and the 15-th is
for 'struct'. Classes with terminals other than the above
16, are attached to the end of 'nd';
p:=initial:=nd(mode(0),NULL);
FOR i:=1 UNTIL 7 DO
  BEGIN nxtmd(p):=nd(mode(i),NULL);
  flag(mode(i)):=bitstring(i); p:=nxtmd(p) END;
FOR i:=1 UNTIL 4 DO
  BEGIN nxtmd(p):=nd(mode(i+11),NULL);
  flag(mode(i)):=bitstring(i+7); p:=nxtmd(p) END;
nxtmd(p):=nd(mode(10),NULL);
flag(mode(10)):=bitstring(12); pre:=p:=nxtmd(p);
nxtmd(p):=nd(mode(11),NULL);
flag(mode(11)):=bitstring(13); p:=nxtmd(p);
nxtmd(p):=nd(mode(9),NULL);
flag(mode(9)):=bitstring(14); p:=nxtmd(p);
nxtmd(p):=nd(mode(8),NULL);
flag(mode(8)):=bitstring(15); final:=p:=nxtmd(p);
class:=15; FOR i:=min UNTIL max DO
  BEGIN IF nf(mode(i))=0 THEN p:=initial ELSE p:=pre;
  WHILE nxtmd(p)¬=NULL DO
    BEGIN IF t(mode(i))¬=t(v(p)) THEN p:=nxtmd(p) ELSE
      BEGIN join_class(mode(i),p); p:=nd(NULL,NULL) END
    END;
  IF v(p)¬=NULL THEN
    BEGIN IF t(mode(i))=t(v(p)) THEN join_class(mode(i),p)
    ELSE
```

```
      BEGIN class:=class+1; nxtmd(p):=nd(mode(i),NULL);
      flag(mode(i)):=bitstring(class) END END END;
  final:=pre; WHILE nxtmd(final)¬=NULL DO
    final:=nxtmd(final);
  COMMENT --- classes are refined by the 'nf' field and the
  class numbers in the flag-field(s) of the field-fields;
  pass_1: pass(1);
  COMMENT --- modes are separated by 'nf' field;
  pass_2: svcl:=0; WHILE svcl<class DO pass(2);
  COMMENT --- modes are separated by the 'class-number' of
  their field or fields;
  COMMENT --- collect the garbage;
    BEGIN REFERENCE(nd) p, q; REFERENCE(md) r, s; INTEGER j;
    COMMENT --- make all the 'link' fields of an 'md' point
    to the first element of the equivalence class and reset
    the 'flag's;
    p:=initial; WHILE v(p)¬=NULL DO
      BEGIN r:=v(p); WHILE r¬=NULL DO
        BEGIN s:=r; flag(r):=#0; r:=link(r); link(s):=v(p)
        END; p:=nxtmd(p) END;
    COMMENT --- go through all the modes and update the
    'field' of the 'md' to point to the unique element
    of the equivalence class;
    p:=initial; WHILE v(p)¬=NULL DO
      BEGIN IF nf(v(p))>0 THEN
      IF field(v(p)) IS md THEN
        field(v(p)):=link(field(v(p))) ELSE
        BEGIN q:=field(v(p)); WHILE q¬=NULL DO
          BEGIN v(q):=link(v(q)); q:=nxtmd(q) END END;
      p:=nxtmd(p) END updating;
    COMMENT --- compact the mode grammar;
    j:=min; FOR i:=min UNTIL max DO
    IF link(mode(i))=mode(i) THEN IF t(mode(i))¬=" " THEN
      BEGIN mode(j):=mode(i); mn(mode(j)):=j; j:=j+1 END;
    FOR i:=j UNTIL max DO
    mode(i):=md(" "," ",#0,i,0,NULL,NULL);
    max:=j-1 END garbage_collection;
  COMMENT --- look at all unions to determine whether they
  satisfy the condition regarding related modes;
  FOR i:=min UNTIL max DO
    IF t(mode(i))="union" THEN
    BEGIN REFERENCE(nd) p, q; q:=NULL;
    COMMENT --- sort and remove repeated modes;
    p:=field(mode(i)); WHILE p¬=NULL DO
      BEGIN searchin(v(p),q); p:=nxtmd(p) END;
    field(mode(i)):=q;  q:=related(q); clean_flag;
    IF q¬=NULL THEN
      BEGIN write("error in mode"); save_md(mode(i));
      save("modes"); savemdlist(q); save("are");
      save("related"); saveout END END END equivalence;

COMMENT --- 'related' determines whether the set of modes in
'list' contains a pair of related modes and if so, delivers
a list of modes of which each is related to some other mode
```

```
of the list;
REFERENCE(nd) PROCEDURE related(REFERENCE(nd) VALUE list);
  BEGIN REFERENCE(nd)p, return; REFERENCE(md)x;
  p:=list; return:=NULL; WHILE p¬=NULL DO
    BEGIN flag(v(p)):=#1; p:=nxtmd(p) END;
  p:=list; WHILE p¬=NULL DO
    BEGIN x:=v(p); WHILE field(x)¬=NULL DO
    IF (t(x)="ref") OR (t(x)="proc") THEN
      BEGIN x:=field(x); IF flag(x)=#1 THEN
        BEGIN searchin(x,return); searchin(v(p),return) END
      END ELSE x:=mode0;
    p:=nxtmd(p) END;
  IF trace_it THEN
    BEGIN save("%related:"); savemdlist(list);
    saveout END;
  return END related;

COMMENT --- 'coerce' determines whether the mode 'mr' can be
'sort'ly coerced to mode 'ms';
LOGICAL PROCEDURE coerce( REFERENCE(md) VALUE mr;
                          REFERENCE(md,nd) VALUE RESULT ms;
                          STRING(6) VALUE RESULT sort;
                          STRING(6) VALUE coercend);
  BEGIN LOGICAL return; REFERENCE(md) mm, m, mq; mq:=m:=mr;
  return:=TRUE; IF sort=" " THEN
    BEGIN REFERENCE(md) mp; mp:=mm:=mr; WHILE mm¬=NULL DO
      BEGIN sort:=IF t(mm)(0|3)="row" THEN "weak" ELSE
                  IF t(mm)="procp" THEN "meek" ELSE " ";
      IF sort=" " THEN
        BEGIN mp:=mm; mm:=field(mm) END ELSE
        BEGIN mq:=IF t(mp)="ref" THEN mp ELSE mm;
        mm:=ms:=NULL END END END;
  IF sort=" " THEN
    BEGIN write ("*** error: dubious sort ***");
    return:=FALSE END ELSE
    BEGIN IF ms=NULL THEN
      BEGIN saveopnd(mr); save(sort); saveon("ly");
      save("coerceable"); save("to"); IF sort¬="strong"
      THEN ms:=possible(nd(mq, NULL), sort);
      savemdlist(ms) END ELSE
      BEGIN save(sort); save("coercion"); save("of");
      save(coercend); saveopnd(mr); save("to");
      saveopnd(ms); save(":");
      IF (sort¬="strong") AND (sort¬="firm") THEN
        BEGIN mm:=mr;
        WHILE (mr¬=ms) AND (mm¬=NULL) DO
        IF fit(mr, sort) THEN
          BEGIN save("de"); saveon(t(mr)(0|4));
          mr:=field(mr); save("to");
          save(mpr(mn(mr))) END ELSE
          BEGIN return:=FALSE; mm:=NULL END END ELSE
        BEGIN WHILE (t(m)="ref") OR (t(m)="proc") DO
          BEGIN flag(m):=¬#1; m:=field(m) END;
        flag(m):=¬#1;
```

```
          IF (flag(ms)=¬#1) AND (t(m)¬="void") THEN
          flag(ms):=#1 ELSE
          post(mr,ms,#1,sort,coercend);
        m:=mr; WHILE (flag(m) AND #1 = #0) AND return DO
        IF (t(m)="ref") OR (t(m)="proc") THEN
          BEGIN save("de"); saveon(t(m)); save("to");
          m:=field(m); save(mpr(mn(m))) END ELSE
        return:=FALSE END END;
    IF return=TRUE THEN WHILE link(m)¬=NULL DO
      BEGIN save(cw(m)); m:=link(m);
      IF link(m)¬=NULL THEN
        BEGIN save("to"); save(mpr(mn(m))) END END
    END; return END coerce;

COMMENT --- 'op_ident' is the operator identification
algorithm of Zosel. It receives a list of eligible operators
and delivers a list of those which can be identified . This
resulting list may have zero, one or more than one element;
BITS PROCEDURE op_ident(REFERENCE(ndo) VALUE eligibleoplist;
                        REFERENCE(md) VALUE r_opnd, l_opnd);
  BEGIN BITS a; BITS ARRAY s_matrix, f_matrix (0::max);

  COMMENT --- 'operand_ident' delivers a bit string that
  identifies the left parameters (i=1) or the right
  parameters (i=2) of the given operators 'eligibeoplist'
  by the operand 'opnd';
  BITS PROCEDURE operand_ident(REFERENCE(ndo) VALUE
      eligibleoplist; INTEGER VALUE i; REFERENCE(md) VALUE
                                  opnd);
    BEGIN REFERENCE(nd)p; BITS b;

    COMMENT --- 'make_pars' delivers an 'nd' list of the
    left (i=1) or right (i=2) parameters of the elements of
    'list';
    REFERENCE(nd)PROCEDURE make_pars(REFERENCE(ndo)VALUE
                                     list; INTEGER VALUE i);
      IF list=NULL THEN NULL ELSE CASE i OF
      (nd(l_par(list),make_pars(nxtop(list),i)),
       nd(r_par(list),make_pars(nxtop(list),i)));

    p:=make_pars(eligibleoplist,i);
    form_matrix(p,#0,#1,"firm",f_matrix);
    IF link(opnd) IS nd THEN
      form_matrix(p,check_hip(link(opnd)),#1,"strong",
                  s_matrix);
      b:=which(p,opnd,"firm",s_matrix,f_matrix);
      b END operand_ident;

  a:=operand_ident(eligibleoplist,2,r_opnd);
  IF l_opnd ¬= NULL THEN
    BEGIN clean_flag; IF a¬=#0 THEN
    a:=a AND operand_ident(eligibleoplist,1,l_opnd) END;
  a END op_ident;
```

```
COMMENT --- 'form_matrix' forms the 'sort'_matrix of the
'plist';
procedure form_matrix(reference(nd) value plist; bits value
                      k,g1; STRING(6) VALUE sort; BITS ARRAY
                      sort_matrix(*) );
  BEGIN REFERENCE(nd) p; BITS f1; p:=plist; f1:=g1;
  WHILE p¬=NULL DO
    BEGIN IF v(p)¬=mode0 THEN
      BEGIN post(mode(1), v(p), f1, sort, " ");
      IF sort="strong" THEN
        BEGIN IF k AND #1 ¬= #0 THEN
        post(mode(skip),v(p),f1,sort," ");
        IF k AND #2 ¬= #0 THEN
        post(mode(jump),v(p),f1,sort," ");
        IF k AND #4 ¬= #0 THEN
        post(mode(nil),v(p),f1,sort," ");
        IF  k AND #8 ¬= #0 THEN
        post(mode(vacuum),v(p),f1,sort," ") END END;
    p:=nxtmd(p); f1:=f1 SHL 1 END;
  FOR i:=0 UNTIL max DO sort_matrix(i):=flag(mode(i));
  IF trace_it THEN
    BEGIN save("%form_matrix"); savemdlist(plist);
    save(sort); saveout END
  END form_matrix;

COMMENT --- 'check_hip' gives a bit string such that each of
the lowest 4 bits is on if 'skip', 'jump', 'nil' or 'vacuum'
is in the 'olist' respectively;
BITS PROCEDURE check_hip(REFERENCE(nd) VALUE olist);
  BEGIN BITS k; REFERENCE(nd) p; k:=#0; p:=olist;
  WHILE p¬=NULL DO
    BEGIN IF v(p) ¬= mode0 THEN
      BEGIN IF ¬(link(v(p)) IS nd) THEN
        BEGIN IF t(v(p))="skip" THEN k:=k OR #1 ELSE
        IF t(v(p))="jump" THEN k:=k OR #2 ELSE
        IF t(v(p))="nil" THEN k:= k OR #4 ELSE
        IF t(v(p))="vacuum" THEN k:= k OR #8 END ELSE
      k:=k OR check_hip(link(v(p))) END;
    p:=nxtmd(p) END;
  k END check_hip;

COMMENT --- 'which' accepts a list of modes 'list' and an
operand 'x' and, depending upon whether 'x' is a mode, a
collateral mode display or a serial/conditional mode
display, it determines which method to use. it returns a bit
string that can be used for selecting those elements of
'list' which can be 'sort'ly coerced from 'x';
BITS PROCEDURE which(REFERENCE(nd) VALUE list;
                     REFERENCE(md) VALUE x; STRING(6) VALUE
                     sort; BITS ARRAY strong_matrix,
                     feat_matrix (*));
  BEGIN BITS b;
  IF trace_it THEN
    BEGIN save("%which:"); savemdlist(list); saveon(";");
```

```
    saveopnd(x); saveon(";");save(sort); saveout END;
  clean_up;   b:= IF ¬(link(x) IS nd) THEN
    BEGIN  IF  sort="strong" THEN id(x,"strong",
                                     strong_matrix) ELSE
    id(x, sort, feat_matrix) END ELSE
  IF cw(x) = "coll" THEN
  collateral(list, link(x), sort) ELSE
  bala(list, link(x), sort,strong_matrix,feat_matrix);
  b END which;

COMMENT --- 'id' delivers a bit string b such that 'x' is
'sort'ly coerceable to each of the modes in the list
corresponding to a '1' of b;
BITS PROCEDURE id(REFERENCE(md) VALUE x; STRING(6) VALUE
                  sort; BITS ARRAY sort_matrix(*));
  BEGIN BITS b; b:=sort_matrix(mn(x));
  IF trace_it THEN
    BEGIN save("%id"); save_md(x); save(sort);
    saveout END;
  WHILE fit(x,sort) DO
    BEGIN x:=field(x); b:=b OR sort_matrix(mn(x)) END;
  b END id;

COMMENT --- 'collateral' delivers a bit string which may be
used for selecting those elements of 'parameterlist' to
which the collateral mode display 'operandlist' may be
'sort'ly balanced;
BITS PROCEDURE collateral(REFERENCE(nd) VALUE RESULT
                          parameterlist; REFERENCE(nd) VALUE
                          operandlist; STRING(6) VALUE
                          sort);
  BEGIN

  COMMENT --- 'fieldlister' delivers a list of the i-th
  field of each of the modes in 'list';
  REFERENCE(nd) PROCEDURE fieldlister(
              INTEGER VALUE i; REFERENCE(nd) VALUE list);
    IF list=NULL THEN NULL ELSE
    IF field(v(list))=NULL THEN nd(mode0,fieldlister(i,
              nxtmd(list))) ELSE
    IF field(v(list)) IS md THEN
      nd(field(v(list)), fieldlister(i,nxtmd(list))) ELSE
      BEGIN REFERENCE(nd) p; p:=field(v(list));
      FOR j:=2 UNTIL i DO p:=nxtmd(p);
      nd(v(p),fieldlister(i,nxtmd(list))) END;

  COMMENT --- 'pick' forms a new 'nd' list such that a mode
  corresponding to a '1' in 'a' is preserved otherwise mode0
  is used;
  reference(nd) procedure pick(bits value a ;
                REFERENCE(nd) VALUE list; INTEGER VALUE i);
    IF list=NULL THEN NULL ELSE
      BEGIN REFERENCE (md) m; m:=v(list);
      nd( IF a AND #1 =#1 THEN
```

```
      CASE i OF ( v(list),
        md(t(m),cw(m),flag(m),mn(m),nf(m),
            pick(¬#0,link(m),i),field(m) ) ) ELSE mode0,
        pick(a SHR 1, nxtmd(list), i) ) END;

BITS f,a,b,b1,row,struct;REFERENCE(nd) plist,rlist,
qlist,p; f:=#1; row:=struct:=#0;
p:=parameterlist; WHILE p¬=NULL DO
  BEGIN IF t(v(p))(0|3)="row" THEN row:=row OR f ELSE
  IF t(v(p))="struct" THEN struct:=struct OR f;
  p:=nxtmd(p); f:=f SHL 1 END;
COMMENT --- consider those which begin with 'row' or
'rowof';
if parameterlist=null then
  BEGIN parameterlist:=nd(mode(9),NULL);
  FOR i:=min UNTIL max DO
  IF t(mode(i))(0|3)="row" THEN
  searchin(mode(i),parameterlist);
  plist:=fieldlister(1,parameterlist) END ELSE
plist:=fieldlister(1,pick(row,parameterlist,1) );
IF trace_it THEN
  BEGIN save("%collateral:"); savemdlist(parameterlist);
  saveon(";"); savemdlist(operandlist); saveon(";");
  save(sort); saveout END;
qlist:=pick(¬#0,operandlist,2);
b:=balance(plist,qlist,sort);
COMMENT --- 'b' remembers those which are possible if we
have a row display. Next consider strong structure
displays;
IF sort="strong" THEN IF struct ¬=#0 THEN
  BEGIN INTEGER no;
  COMMENT --- first count the number 'no' of elements in
  the structure display 'operandlist';
  p:=operandlist; no:=0; WHILE p¬=NULL DO
    BEGIN no:=no+1; p:=nxtmd(p) END;
  b1:=#0; f:=#1; p:=parameterlist;
  WHILE p¬=NULL DO
    BEGIN IF nf(v(p))=no THEN b1:=b1 OR f;
    p:=nxtmd(p); f:=f SHL 1 END;
  COMMENT --- eliminate those structures of incompatible
  length;
  b1:=b1 AND struct; IF b1¬=#0 THEN
  BEGIN plist:=pick(b1,parameterlist,1);
  f:=#1; p:=parameterlist; a:=b1;
  FOR i:=1 UNTIL no DO
  IF a¬=#0 THEN
      BEGIN COMMENT --- consider each operand element in
            turn;
      BITS ARRAY strong_matrix(0::max); BITS k;
      rlist:=fieldlister(i, plist);
      COMMENT --- 'rlist' contains the corresponding
      portrayals;
      p:=rlist; WHILE p¬=NULL DO
        BEGIN COMMENT --- jump over the selector to get
```

```
                the mode;
          v(p):= IF field(v(p))=NULL THEN mode0 ELSE
                 field(v(p)); p:=nxtmd(p) END;
        clean_flag; IF link(v(operandlist)) IS nd THEN
        k:=check_hip(link(v(operandlist))) ELSE
        k:=check_hip(nd(v(operandlist),NULL));
        form_matrix(rlist,k,#1,"strong",strong_matrix);
        a:=a AND which(rlist,v(operandlist),"strong",
                       strong_matrix,strong_matrix);
        operandlist:=nxtmd(operandlist) END;
      b:=b OR a END END;
  b END collateral;

COMMENT --- 'balance' produces a bit string which may be
used for selecting those elements of 'parameterlist' to
which the serial/conditional display 'operandlist' may be
'sort'ly balanced. Note that if 'parameterlist' is null and
the 'sort' is meek, weak or soft, then it will supply its
own by calling 'possible'. If the sort is not given, then it
will check if the 'operandlist' is a call or a slice, and it
will supply the suitable 'sort' and the 'parameterlist';
BITS PROCEDURE balance(REFERENCE(nd) VALUE RESULT
                       parameterlist, operandlist;
                       STRING(6) VALUE RESULT sort);
  BEGIN REFERENCE(nd) plist, p; BITS a;
  BITS ARRAY strong_matrix, feat_matrix (0::max);
  ravel(operandlist, 2);
  IF sort=" " THEN
    BEGIN REFERENCE(nd) mm; REFERENCE(md) mp, mq;
    mm:=operandlist; WHILE mm¬=NULL DO
      BEGIN mp:=v(mm);
      IF (t(mp)="skip") OR (t(mp)="jump") OR (t(mp)="nil")
        OR (t(mp)="vacuum") THEN
      mm:=nxtmd(mm) ELSE mm:=NULL END;
    mq:=mp; WHILE mp¬=NULL DO
      BEGIN sort := IF t(mp)(0|3) = "row" THEN "weak" ELSE
                    IF t(mp) = "procp" THEN "meek" ELSE " ";
      IF sort = " " THEN
        BEGIN mq:=mp; mp:=field(mp) END ELSE
        BEGIN IF t(mq)¬="ref" THEN mq:=mp;
        parameterlist:=possible(nd(mq, NULL), sort)
        END END END;
  IF sort=" " THEN
    BEGIN write("*** error: dubious sort ***" ); a:=#0 END
  ELSE
    BEGIN IF parameterlist = NULL THEN
    IF (sort¬="strong") AND (sort¬="firm") THEN
    parameterlist:=possible(operandlist, sort);
    plist:=parameterlist;
    form_matrix(plist, check_hip(operandlist), #1, "strong",
                strong_matrix);
    IF sort ¬= "strong" THEN
      BEGIN clean_flag;
      form_matrix(plist,#0,#1,sort,feat_matrix) END;
```

```
    a:=bala(plist,operandlist,sort,strong_matrix,
            feat_matrix);
    IF trace_it THEN
      BEGIN save("%balance:"); savemdlist(parameterlist);
      saveon(";"); savemdlist(operandlist); saveon(";");
      save(sort); saveout END END;
  a END balance;

COMMENT --- 'bala' yields a bit string such that each of the
corresponding modes in the 'plist' can be 'sort'ly balanced
from the 'olist';
BITS PROCEDURE bala(REFERENCE(nd) VALUE plist, olist;
                    STRING(6) VALUE sort; BITS ARRAY
                    strong_matrix, feat_matrix (*));
  BEGIN BITS a,b; REFERENCE(nd) p;
  p:=olist; a:=¬#0; b:=#0;
  WHILE p¬=NULL DO
    BEGIN IF v(p)¬=mode0 THEN
      BEGIN a := a AND which(plist, v(p), "strong",
                             strong_matrix, feat_matrix);
      IF sort ¬= "strong" THEN
      b := b OR which(plist, v(p), sort, strong_matrix,
                      feat_matrix) END;
    p := nxtmd(p) END;
  IF trace_it THEN
    BEGIN save("%bala"); savemdlist(olist);
    save(sort); saveout END;
  IF sort="strong" THEN a ELSE a AND b END bala;

COMMENT --- 'test_them' tests all the procedures;
PROCEDURE test_them;
    BEGIN

  COMMENT --- 'reverse_ravel' is the reverse of raveling
  unions. For each union mode, add to its field all defined
  modes which are its proper subunions. A proper subunion
  of a given union mode is a union, the set of whose
  constituent modes is a proper subset of that of the given
  mode.;
  PROCEDURE reverse_ravel;
    BEGIN

    COMMENT --- 'include' determines whether the sorted 'nd'
    list 'p' is included in the sorted 'nd' list 'q';
    LOGICAL PROCEDURE include(REFERENCE(nd) VALUE p, q);
      IF p=NULL THEN TRUE ELSE IF q=NULL THEN FALSE ELSE
      IF v(p)=v(q) THEN include(nxtmd(p),nxtmd(q)) ELSE
      include(p,nxtmd(q));

    FOR i:=min UNTIL max DO
      BEGIN REFERENCE(md) mi; mi:=mode(i);
      IF t(mi)="union" THEN
        BEGIN FOR j:=min UNTIL max DO
          BEGIN IF t(mode(j))="union" THEN
```

```
        IF i¬=j THEN
        IF include(field(mode(j)), field(mi)) THEN
          BEGIN REFERENCE(nd) mq; mq:=field(mi);
          WHILE nxtmd(mq)¬=NULL DO mq:=nxtmd(mq);
          nxtmd(mq):=nd(mode(j),NULL) END END END END;
  IF trace_it THEN
    BEGIN save("%reverse_ravel"); saveout END
  END reverse_ravel;

REFERENCE(nd) md11, md12; REFERENCE(md) lo;
REFERENCE(md,nd) ro; REFERENCE(ndo) op; STRING test;
STRING(6) sort, coercend; BITS b;
COMMENT --- perform the required tests;
write(" ");
write("enter command: don't forget quotes"); write(" ");
read(test); write("     "); writeon(test);
IF test="trace" THEN
  BEGIN write("trace turned on"); trace_it:= TRUE END ELSE
IF test="notrace" THEN
  BEGIN trace_it:=FALSE; write("trace turned off") END
ELSE IF test="equivalence" THEN
  BEGIN equivalence;
  FOR i:=0 UNTIL min-1 DO
    BEGIN flag(mode(i)):=#0; link(mode(i)):=NULL END;
  write ("modes equivalenced");
  FOR i:=min UNTIL max DO
    BEGIN save(mpr(i)); save("="); save_md(mode(i));
    saveout; flag(mode(i)):=#0; link(mode(i)):=NULL END;
  reverse_ravel END ELSE
IF test="related" THEN
  BEGIN write("enter: modelist"); write("   ");
  save("mode.list");
  md11:=readmdlist; savemdlist(md11); saveout;
  md12:=related(md11);
  IF md12¬=NULL THEN
    BEGIN save("the"); save("set"); savemdlist(md12);
    save("contains"); save("related"); save("modes.");
    saveout END ELSE
  write("no two of these modes are related");
  clean_flag END ELSE
IF test="coerce" THEN
  BEGIN write("enter: a priori mode, a posteriori mode,");
  writeon(" sort, coercend"); write("   ");
  lo:=readmd; ro:=readmd; readon(sort); readon(coercend);
  writeon(sort); writeon("   "); writeon(coercend);
  IF coerce(lo, ro, sort, coercend) THEN saveout ELSE
    BEGIN out_line:=" "; out_ptr:=1; save_md(lo);
    save("not"); save(sort); saveon("ly");
    save("coerceable"); save("to"); save_md(ro);
    saveout END;
  clean_up END ELSE
IF test="identify" THEN
  BEGIN write("enter: mode, sort, modelist");
  write("   "); ro:=readmd; save("mode:"); saveopnd(ro);
```

```
    saveon(";"); readon(sort); save("sort:"); save(sort);
    saveon(";"); writeon(sort); md11:=readmdlist;
    save("mode.list:"); savemdlist(md11); saveout;
    write("possible modes identified:  ");
    savemdlist(select(identify(md11,ro,sort), md11));
    clean_up; saveout END ELSE
  IF (test="balance") OR (test="collateral") THEN
    BEGIN
    write("enter: operand modes, sort, parameter modes");
    write("   "); md11:=readmdlist;
    save("operand.display:"); savemdlist(md11); saveon(";");
    readon(sort); save("sort:"); writeon(sort);
    save(sort); saveon(";"); md12:=readmdlist;
    save("parameters:"); savemdlist(md12); saveout;
    write("possible ", test, ":");
    b:= IF test="balance" THEN balance(md12,md11,sort) ELSE
                               collateral(md12,md11,sort);
    savemdlist(select(b, md12)); clean_up; saveout END ELSE
  IF test="operators" THEN
    BEGIN write("enter: left operand, right operand,",
                " operators"); write("   ");
    lo:=readmd; ro:=readmd; write("   "); op:=readoplist;
    save("left.operand:"); saveopnd(lo); saveon(";");
    saveout; save("right.operand:"); saveopnd(ro);
    saveon(";"); saveout; save("operators:");
    saveoplist(op); saveout; write("possible operators:");
    saveoplist(selecto(op_ident(op,ro,lo),op)); clean_up;
    saveout END ELSE
  IF test="grammar" THEN enter_grammar ELSE
  IF test="stop" THEN work:=FALSE ELSE
    BEGIN write("sorry - try again",
    " -- coerce, related, grammar, identify, balance,",
    " collateral, operators, equivalence, trace,",
    " notrace or stop."); write(" ") END END test_them;

COMMENT --- 'identify' delivers a bit string that can be
used for selecting those elements of 'plist' to which the
mode 'x' may be 'sort'ly coerced;
BITS PROCEDURE identify(REFERENCE(nd) VALUE plist;
                        REFERENCE(md) VALUE x;
                        STRING(6) VALUE sort);
  BEGIN BITS ARRAY strong_matrix, feat_matrix (0::max);
  BITS k, b; k:=#0;
  IF trace_it THEN
    BEGIN save("%identify:"); savemdlist(plist);
    saveon(";"); saveopnd(x); saveon(";"); save(sort);
    saveout END;
  IF sort="strong" THEN
    BEGIN form_matrix(plist,check_hip(nd(x,NULL)),#1,sort,
                      strong_matrix);
    b:=id(x,sort,strong_matrix) END ELSE
    BEGIN form_matrix(plist,#0,#1,sort,feat_matrix);
    b:=id(x,sort,feat_matrix) END;
  b END identify;
```

```
COMMENT --- 'selecto' does the same job as 'select' but for
'ndo'-lists;
REFERENCE(ndo)PROCEDURE selecto(BITS VALUE a;
                                REFERENCE(ndo) VALUE list);
  IF list=NULL THEN NULL ELSE IF a AND #1 = #1 THEN
  ndo(op_symbol(list), l_par(list), r_par(list),
      selecto(a SHR 1, nxtop(list))) ELSE
  selecto(a SHR 1, nxtop(list));

COMMENT --- 'enter_grammar' accepts a number of rules of the
form 'n s m', where 'n' is an integer, 's' is a string and
'm' is an integer (representing one mode) or an integer
sequence followed by -2 (representing a list of modes). The
list is used in the case that the string is "struct",
"union" or "procp";
PROCEDURE enter_grammar;
  BEGIN INTEGER i; STRING(6) terminal; max:=15;
  FOR i:=min UNTIL max+20 DO
  mode(i):=md(" ", " ", #0, i, 0, NULL, NULL);
  write("enter the grammar:"); write("     ");
  WHILE BEGIN read(i); write("   "); writeon(i); i>-2 END DO
    BEGIN IF (i>max) AND (i<35) THEN max:=i;
    IF (i<min) OR (i>34) THEN
      BEGIN i:=35; IF t(mode(35))=" " THEN
      write("*** warning *** there is a restriction --- ",
            "15<i<35. ") ELSE
      write("*** i is outside the limits 15<i<35 again, ",
            "the previous mode(i) with i outside the ",
            "limits will not be entered ***") END;
    readon(terminal); t(mode(i)):=terminal; writeon("   ");
    writeon(terminal); field(mode(i)):=
      IF (terminal ¬= "union") AND
         (terminal ¬= "struct") AND
         (terminal ¬= "procp") THEN readmd ELSE
      BEGIN INTEGER n; REFERENCE(nd)l1,l2; l1:=readmdlist;
      l2:=l1; n:=0; WHILE l2¬=NULL DO
        BEGIN l2:=nxtmd(l2); n:=n+1 END;
      nf(mode(i)):=n;  l1 END;
    IF field(mode(i)) IS md THEN
      nf(mode(i)):=1 END;
  IF t(mode(35))¬=" " THEN
    BEGIN IF max<35 THEN
      BEGIN max:=max+1; mode(max):=mode(35) END END;
  COMMENT --- print the mode table, first as a grammar and
  second as a list of individual modes;
  FOR i:=min UNTIL max DO
  IF mode(i) ¬= NULL THEN
  IF field(mode(i)) ¬= NULL THEN
    BEGIN REFERENCE(md)m; save(mpr(i)); save("=");
    m:=mode(i); save(t(m)); IF field(m) IS md THEN
    save(mpr(mn(field(m)))) ELSE
      BEGIN REFERENCE(nd) p; p:=field(m); WHILE p¬=NULL DO
        BEGIN save(mpr(mn(v(p)))); p:=nxtmd(p); IF p¬=NULL
```

```
        THEN saveon(",") END END;
    saveout END;
  write(" "); FOR i:=min UNTIL max DO
    BEGIN save(mpr(i)); save("="); save_md(mode(i));
    saveout END END enter_grammar;

COMMENT --- 'readmdlist' reads a sequence of integers which
it interpretes as a list of modes. The list must end with
the value '-2';
REFERENCE(nd) PROCEDURE readmdlist;
  BEGIN REFERENCE(md) m; m:=readmd;
  IF m=NULL THEN NULL ELSE nd(m, readmdlist) END readmdlist;

COMMENT --- 'readmd' reads either a mode (i.e., an integer)
or a mode display (i.e. -1 followed by a string followed by
an integer sequence followed by -2). Note that during input
of the grammar this procedure is always given a non-negative
integer;
reference(md) procedure readmd;
  BEGIN INTEGER i; REFERENCE(md) m; readon(i); writeon(i);
  IF i>=0 THEN m:=mode(i) ELSE IF i<=-2 THEN m:=NULL ELSE
    BEGIN STRING(6) s; readon(s); writeon(s);
    m:=md(" ", s, #0, -1, 0, readmdlist, NULL) END;
  m END readmd;

COMMENT --- 'readoplist' reads a sequence of integer pairs
which are interpreted as the left and right parameter modes
of the operator '+'. It is terminated by the pair -2 -2;
REFERENCE(ndo) PROCEDURE readoplist;
  BEGIN REFERENCE(md)m1,m2; m1:=readmd; m2:=readmd;
  IF m2=NULL THEN NULL ELSE
  ndo("+", m1, m2, readoplist) END readoplist;

COMMENT --- 'saveoplist' saves the operator list 'ol' for
output;
PROCEDURE saveoplist(REFERENCE(ndo) VALUE ol);
  WHILE ol¬=NULL DO
    BEGIN save("("); saveopnd(l_par(ol));
    saveon(op_symbol(ol)); saveopnd(r_par(ol)); saveon(")");
    ol:=nxtop(ol);
    IF ol¬=NULL THEN saveon(",") END saveoplist;

COMMENT --- 'saveopnd' saves a mode or a mode display for
output;
PROCEDURE saveopnd(REFERENCE(md) VALUE lo);
  IF lo=NULL THEN save("null") ELSE
    IF link(lo) IS nd THEN
    BEGIN save("("); savemdlist(link(lo));saveon(")") END
  ELSE save_md(lo);

COMMENT --- 'savemdlist' saves the mode list 'mdl' for
output;
PROCEDURE savemdlist(REFERENCE(nd) VALUE mdl);
  WHILE mdl¬=NULL DO
```

```
    BEGIN saveopnd(v(md1)); md1:=nxtmd(md1);
    IF md1¬=NULL THEN saveon(",") END savemdlist;

PROCEDURE saveon(STRING VALUE s1);
  BEGIN out_ptr:=out_ptr-1; save(s1) END saveon;

COMMENT --- 'save' saves the non-blank part of 's1' in
'out_line' followed by one blank;
PROCEDURE save(STRING VALUE s1);
  BEGIN INTEGER i; i:=0; IF out_ptr>225 THEN saveout;
  WHILE i<=15 DO
    IF s1(i|1) = " " THEN i:=16 ELSE
      BEGIN out_line(out_ptr|1):=s1(i|1); i:=i+1;
      out_ptr:=out_ptr+1 END;
  out_ptr:=out_ptr+1 END save;

COMMENT --- 'save_md' saves the mode 'm' in the string
'out_line' for the purpose of readable output;
PROCEDURE save_md(REFERENCE(md) VALUE m);
  BEGIN IF m ¬= NULL THEN
    BEGIN BITS sgn; sgn:=#80000000;
    IF nf(m)=0 THEN save(t(m)) ELSE
    IF flag(m) AND sgn=#0 THEN
      BEGIN save(t(m)); flag(m):=flag(m) OR sgn;
      IF (t(m)¬="union") AND (t(m)¬="procp") AND
         (t(m)¬="struct") THEN
      save_md(field(m)) ELSE
      IF t(m)="procp" THEN
        BEGIN REFERENCE(nd) p; save("(");
        out_ptr:=out_ptr-1; p:= field(m); WHILE p¬=NULL DO
          BEGIN IF nxtmd(p)=NULL THEN saveon(")");
          save_md(v(p)); p:=nxtmd(p) END END ELSE
        BEGIN save("("); savemdlist(field(m));
        saveon(")") END;
      flag(m):=flag(m) AND ¬sgn END ELSE
    save(mpr(mn(m))) END END save_md;

COMMENT --- 'saveout' prints out the output buffer
'out_line';
PROCEDURE saveout;
  BEGIN write(out_line(0|60)); IF out_ptr>60 THEN
  write(out_line(60|60));
  IF out_ptr>120 THEN write(out_line(120|60));
  IF out_ptr>180 THEN write(out_line(180|60));
  out_ptr:=1; out_line:=" " END saveout;

LOGICAL trace_it, work;
INTEGER void, int, bool, real1, char, format, bits1, bytes,
  strng, compl, skip, jump, nil, vacuum, min, max, out_ptr;
REFERENCE(md) ARRAY mode(0::35); REFERENCE(md) mode0;
STRING(240) out_line;

COMMENT --- initialization part. 'trace_it' is a logical
variable which controls some built in tracing;
```

```
start: intfieldsize:=4; out_line:=" "; out_ptr:=1;
trace_it:=FALSE; work:=TRUE; void:=0; bool:=1; int:=2;
real1:=3; char:=4; format:=5; bits1:=6; bytes:=7; compl:=8;
strng:=9; skip:=12; jump:=13; nil:=14; vacuum:=15;
min:=16; max:=15;
mode0:=md(" "," ",#0,-1,0,NULL,NULL);
mode(void):=md("void"," ",#0,0,0,NULL,NULL);
mode(bool):=md("bool"," ",#0,1,0,NULL,NULL);
mode(int):=md("int"," ",#0,2,0,NULL,NULL);
mode(real1):=md("real"," ",#0,3,0,NULL,NULL);
mode(char):=md("char"," ",#0,4,0,NULL,NULL);
mode(format):=md("format"," ",#0,5,0,NULL,NULL);
mode(bits1):=md("bits"," ",#0,6,0,NULL,NULL);
mode(bytes):=md("bytes"," ",#0,7,0,NULL,NULL);
mode(10):=md("re"," ",#0,10,1,NULL,mode(real1));
mode(11):=md("im"," ",#0,11,1,NULL,mode(real1));
mode(skip):=md("skip"," ",#0,12,0,NULL,NULL);
mode(jump):=md("jump"," ",#0,13,0,NULL,NULL);
mode(nil):=md("nil"," ",#0,14,0,NULL,NULL);
mode(vacuum):=md("vacuum"," ",#0,15,0,NULL,NULL);
mode(compl):=md("struct"," ",#0,8,2,NULL,
     nd(mode(10),nd(mode(11),NULL)));
mode(strng):=md("rowof"," ",#0,9,1,NULL,mode(char));
WHILE work DO test_them;
END.
```

How To Use The Program

The program accepts the following commands which must be enclosed in quotes:
"TRACE", "NOTRACE", "GRAMMAR", "EQUIVALENCE", "COERCE", "RELATED", "IDENTIFY", "BALANCE", "COLLATERAL", "OPERATORS" and "STOP".
Except for "TRACE", "NOTRACE", "EQUIVALENCE" and "STOP", input in a certain format is to be entered. A mode which is not null is represented by a non-negative integer less than or equal to the maximum of the rule number entered in the grammar (that is less than or equal to 35). A null mode is represented by an integer less than -1. A mode list is a sequence of modes ($0 \leq$ integers ≤ 35) followed by an integer less than -1. A mode display is a mode list preceded by -1 "coll", -1 "seri" or -1 "cond" e.g. -1 "ccll" 2 3 14 -2. 'sort' is a string, it is "strong", "firm", "meek", "weak" or "soft". 'coercend' is a string, it is either "comorf" to tell that the coercend is a comorf or "morf" (or " ") to tell that the coercend is not a comorf. An operand is a mode or a mode display. A list of operators is a sequence of pairs of operands terminated by -2 -2. An unary operator is an operator whose first (left) operand is null.

The standard modes of the program are as follows:

```
m0 = void
m1 = bool
m2 = int
m3 = real
m4 = char
m5 = format
m6 = bits
m7 = bytes
m8 = struct m10 m11 (complex)
m9 =  rowof m4      (string)
m10 = re m3
m11 = im m3
m12 = skip
m13 = jump
m14 = nil
m15 = vacuum.
```

(1) The command "TRACE" turns on a built in trace, "NOTRACE" turns that off.

(2) When the command is "GRAMMAR", rules of a mode grammar followed by an integer less than -1 that shows the end of the set of rules, are to be entered. Each rule of the mode grammar is entered as

m t p

where

(a) m is an integer such that 15 < m < 35.

(b) t is a terminal which is a string so that it must be enclosed in quotes. The terminals are "ref", "proc", "row", "rowof", "union", "struct", "procp" (procedure with parameters), or a field-selector of a structure, or any other standard mode.

(c) p is either a mode or when t is "struct", "union" or "procp" a mode list.

This command allocates a record 'md' to each rule entered, so that they can be used by some other commands to follow.

(3) The command "EQUIVALENCE" compacts the mode grammar by removing equivalent modes.

(4) When the command is "COERCE", the source mode(the a priori mode), the target mode(the a posteriori mode), the sort(position) and the coercend are to be entered. It determines whether the a priori mode is sortly coerceable to the a posteriori mode.

(5) For the command "RELATED", the input is a mode list. This determines whether there are related modes in the mode list.

(6) The input following "IDENTIFY" is mode, sort and mode

list. This is to identify from the mode list those modes to which the given mode is sortly coerceable.

(7) The input following "BALANCE" or "COLLATERAL" is source mode list, sort, target mode list. The source mode list contains the modes in a conditional/serial clause or a collateral clause, and any one of these modes may be a serial/conditional or collateral mode display. The command "BALANCE" determines from the target mode list the modes to which the serial/conditional mode display (i.e. the source mode list) can be sortly coerced. The command "COLLATERAL" determines from the target mode list those modes to which the collateral mode display can be sortly coerced.

(8) The input following the command "OPERATORS" is left operand, right operand and the list of operators. This identifies from the list of operators that operator (or those operators if ambiguous) identified by a formula with the given operands. Note that unary operators and binary operators are not to be mixed in the operators and that only a list of unary operators will be given as a target list for a monadic formula.

(9) The command "STOP" terminates the execution.

A Sample Run

```
ENTER COMMAND: DON'T FORGET QUOTES

     NOTRACE
TRACE TURNED OFF

ENTER COMMAND: DON'T FORGET QUOTES

     GRAMMAR
ENTER THE GRAMMAR:

      16      REF      3
      17      PROC     3
      18      REF     17
      19      ROWOF    3
      20      REF     19
      21      UNION    2     3     19    -2
      22      STRUCT  23    25    -2
      23      A       24
      24      ROWOF    4
      25      B       26
      26      REF     22
      27      ROWOF    1
      28      UNION    1    21    -2
      29      UNION    1    21    -2
      30      REF     30
      -2
```

```
 M16 = REF REAL
 M17 = PROC REAL
 M18 = REF M17
 M19 = ROWOF REAL
 M20 = REF M19
 M21 = UNION INT, REAL, M19
 M22 = STRUCT M23, M25
 M23 = A M24
 M24 = ROWOF CHAR
 M25 = B M26
 M26 = REF M22
 M27 = ROWOF BOOL
 M28 = UNION BOOL, M21
 M29 = UNION BOOL, M21
 M30 = REF M30

 M16 = REF REAL
 M17 = PROC REAL
 M18 = REF PROC REAL
 M19 = ROWOF REAL
 M20 = REF ROWOF REAL
 M21 = UNION ( INT, REAL, ROWOF REAL)
 M22 = STRUCT ( A ROWOF CHAR, B REF M22)
 M23 = A ROWOF CHAR
 M24 = ROWOF CHAR
 M25 = B REF STRUCT ( A ROWOF CHAR, M25)
 M26 = REF STRUCT ( A ROWOF CHAR, B M26)
 M27 = ROWOF BOOL
 M28 = UNION ( BOOL, UNION ( INT, REAL, ROWOF REAL))
 M29 = UNION ( BOOL, UNION ( INT, REAL, ROWOF REAL))
 M30 = REF M30

ENTER COMMAND: DON'T FORGET QUOTES
     EQUIVALENCE
CONTEXT CONDITION ERROR INVOLVING THE MODE
 REF M30,
 WHICH IS REPLACED BY 'BOOL'.
MODES EQUIVALENCED
 M16 = REF REAL
 M17 = PROC REAL
 M18 = REF PROC REAL
 M19 = ROWOF REAL
 M20 = REF ROWOF REAL
 M21 = UNION ( INT, REAL, ROWOF REAL)
 M22 = STRUCT ( A ROWOF CHAR, B REF M22)
 M23 = A ROWOF CHAR
 M24 = B REF STRUCT ( A ROWOF CHAR, M24)
 M25 = REF STRUCT ( A ROWOF CHAR, B M25)
 M26 = ROWOF BOOL
 M27 = UNION ( BOOL, INT, REAL, ROWOF REAL)

ENTER COMMAND: DON'T FORGET QUOTES

     COERCE
```

```
ENTER: A PRIORI MODE, A POSTERIORI MODE, SORT, COERCEND
     18      0  STRONG   COMORF
 STRONG COERCION OF COMORF REF PROC REAL TO VOID : VOIDED TO
 VOID

ENTER COMMAND: DON'T FORGET QUOTES

     COERCE
ENTER: A PRIORI MODE, A POSTERIORI MODE, SORT, COERCEND
     18      0  STRONG   MORF
 STRONG COERCION OF MORF REF PROC REAL TO VOID : DEREF TO M1
7 DEPROC TO REAL VOIDED TO VOID

ENTER COMMAND: DON'T FORGET QUOTES

     COERCE
ENTER: A PRIORI MODE, A POSTERIORI MODE, SORT, COERCEND
     18     21  FIRM     MORF
 FIRM COERCION OF MORF REF PROC REAL TO UNION ( INT, REAL, R
OWOF REAL) : DEREF TO M17 DEPROC TO REAL UNITE TO M21

ENTER COMMAND: DON'T FORGET QUOTES

     COERCE
ENTER: A PRIORI MODE, A POSTERIORI MODE, SORT, COERCEND
     18     21  MEEK     MORF
 REF PROC REAL NOT MEEKLY COERCEABLE TO UNION ( INT, REAL, R
OWOF REAL)

ENTER COMMAND: DON'T FORGET QUOTES

     COERCE
ENTER: A PRIORI MODE, A POSTERIORI MODE, SORT, COERCEND
     18     -2  FIRM     MORF
 REF PROC REAL FIRMLY COERCEABLE TO REAL, PROC REAL, REF PRO
C REAL

ENTER COMMAND: DON'T FORGET QUOTES
     RELATED
ENTER: MODELIST
      3     16     17     18     19     -2
 MODE.LIST REAL, REF REAL, PROC REAL, REF PROC REAL, ROWOF R
EAL
 THE SET REAL, REF REAL, PROC REAL, REF PROC REAL CONTAINS R
ELATED MODES.

ENTER COMMAND: DON'T FORGET QUOTES

     IDENTIFY
ENTER: MODE, SORT, MODELIST
     18  STRONG  16     20     21     22     -2
 MODE: REF PROC REAL; SORT: STRONG; MODE.LIST: REF REAL, REF
 ROWOF REAL, UNION ( INT, REAL, ROWOF REAL), STRUCT ( A ROWO
F CHAR, B REF M22)
```

```
POSSIBLE MODES IDENTIFIED:
 UNION ( INT, REAL, ROWOF REAL)

ENTER COMMAND: DON'T FORGET QUOTES

     BALANCE
ENTER: OPERAND MODES, SORT, PARAMETER MODES
     20    16    -2  WEAK     3    18    19    20    -2
 OPERAND.DISPLAY: REF ROWOF REAL, REF REAL; SORT: WEAK; PARA
METERS: REAL, REF PROC REAL, ROWOF REAL, REF ROWOF REAL
POSSIBLE BALANCE         :
 REF ROWOF REAL

ENTER COMMAND: DON'T FORGET QUOTES

     BALANCE
ENTER: OPERAND MODES, SORT, PARAMETER MODES
     20    18    -2  MEEK     3    18    19    20    -2
 OPERAND.DISPLAY: REF ROWOF REAL, REF PROC REAL; SORT: MEEK;
 PARAMETERS: REAL, REF PROC REAL, ROWOF REAL, REF ROWOF REAL

POSSIBLE BALANCE         :
 ROWOF REAL

ENTER COMMAND: DON'T FORGET QUOTES

     BALANCE
ENTER: OPERAND MODES, SORT, PARAMETER MODES
     17    18    -2  FIRM    -2
 OPERAND.DISPLAY: PROC REAL, REF PROC REAL; SORT: FIRM; PARA
METERS:
POSSIBLE BALANCE         :


ENTER COMMAND: DON'T FORGET QUOTES

     COLLATERAL
ENTER: OPERAND MODES, SORT, PARAMETER MODES
      9    25    -2  STRONG  19    20    21    22    -2
 OPERAND.DISPLAY: ROWOF CHAR, REF STRUCT ( A ROWOF CHAR, B M
25); SORT: STRONG; PARAMETERS: ROWOF REAL, REF ROWOF REAL, U
NION ( INT, REAL, ROWOF REAL), STRUCT ( A ROWOF CHAR, B REF
M22)
POSSIBLE COLLATERAL      :
 STRUCT ( A ROWOF CHAR, B REF M22)

ENTER COMMAND: DON'T FORGET QUOTES

     OPERATORS
ENTER: LEFT OPERAND, RIGHT OPERAND, OPERATORS
     18     3
     21    21    22    22     8     3    -2    -2
 LEFT.OPERAND: REF PROC REAL;
 RIGHT.OPERAND: REAL;
```

```
 OPERATORS: ( UNION ( INT, REAL, ROWOF REAL)+ UNION ( INT, R
EAL, ROWOF REAL)), ( STRUCT ( A ROWOF CHAR, B REF M22)+ STRU
CT ( A ROWOF CHAR, B REF M22)), ( STRUCT ( RE REAL, IM REAL)
+ REAL)
POSSIBLE OPERATORS:
 ( UNION ( INT, REAL, ROWOF REAL)+ UNION ( INT, REAL, ROWOF
REAL))

ENTER COMMAND: DON'T FORGET QUOTES

     OPERATORS
ENTER: LEFT OPERAND, RIGHT OPERAND, OPERATORS
     -2      3
     -2      3    -2     17    -2     16    -2     -2
 LEFT.OPERAND: NULL;
 RIGHT.OPERAND: REAL;
 OPERATORS: ( NULL+ REAL), ( NULL+ PROC REAL), ( NULL+ REF R
EAL)
POSSIBLE OPERATORS:
 ( NULL+ REAL)

ENTER COMMAND: DON'T FORGET QUOTES

     STOP
0001.64 SECONDS IN EXECUTION
```

APPENDIX B

REVISED SYNTAX RULES

This thesis is based on the syntax rules which are brought forward to the meeting at Vienna in September, 1972, (it is not yet adopted). The rules which are modified and concern with this thesis are as follows:
The numbers are the same as used in the Report.

1.2.1. Metaproduction Rules of Modes

o) STOWED: structured with FIELDS; ROWS of MODE.
vb) ROWS: row; ROWS row.

1.2.2. Metaproduction Rules Associated with Modes

bb) ROW: row; row of.
d) deleted.
ea) NONREF: UNITED; PLAIN; format; PROCEDURE; structured with FIELDS; ROWS of MODE.
h) NONPROC: PLAIN; format; procedure with PARAMETERS MOID; reference to NONPROC; UNITED; structured with FIELDS; row of MODE.
ib) PARAMS: parameter and PARAMETERS; parameter.
rb) FOLDS: field TAG and FIELDS; field TAG.

1.2.3. Metaproduction Rules Associated with Phrases and Coercion

a) deleted.
b) deleted.
c) SOME: SORT MOID.
cb) SIGNLE: unitary; ENCLOSED.
d) ENCLOSED: closed; collateral; CHOICE.
e) deleted.
ea) CHOICE: condition; case; conformity.
eb) UNETY: UNITED; EMPTY.
ec) CONFETY: UNITED conformity; EMPTY.
f) deleted.
g) SORT: strong; FEAT.
h) FEAT: firm; meek; weak; soft.
i) STRONG: FIRM; widened; rowed; voided.
l) FIRM: MEEK; united.
lb) MEEK: unchanged from; deprocedured; dereferenced.
o) FROBYT: from; by; to.

1.2.4. Metaproduction Rules Associated with Coercends

b) FORM: MORF; COMORF.
bb) FORMSPEC: FORM; specification.
c) deleted.
ca) MORF: routine text; PRIETY ADIC formula; selection; mode identifier; slice; call.
cb) COMORF: assignation; cast; identity relation;

generator; denotation.
d) ADIC: dyadic; monadic.
eb) PRIETY: PRIORITY; EMPTY.

6.1. Serial Clauses

6.1.1. Syntax

a) SOME serial clause: declaration prologue series option, SOME parade.
b) declaration prologue: strong void unit series option, SINGLE declaration.
c) deleted.
d) deleted.
e) SOME unit: SOME unitary clause.
f) deleted.
g) SORT MOID parade: SORT MOID train; SORT MOID train, completion token, label, strong MOID parade; strong MOID train, completion token, label, SORT MOID parade.
h) SOME train: strong void labelled unit series option, SOME labelled unit.
i) deleted.
j) SOME labelled unit: label sequence option, SOME unit.

6.2. Collateral Phrases

6.2.1. Syntax

c) STIRM ROW MODE collateral clause: STIRM MODE balance PACK.
d) deleted.
e) SORT MOID CONFETY balance: SORT MOID CONFETY unit, comma token, strong MOID CONFETY unit list; strong MOID CONFETY unit, comma token, SORT MOID CONFETY unit; strong MOID CONFETY unit, comma token, SORT MOID CONFETY balance.
f) strong structured with FIELDS and FIELD collateral clause: FIELDS and FIELD portrait PACK.
g) FIELDS and FIELD portrait: FIELDS portrait comma token, FIELD portrait.
h) MODE field TAG portrait: strong MODE unit.

6.3. Closed Clauses

6.3.1. Syntax

a) SOME closed clause: SOME serial clause PACK.

6.4. Choice Clauses

6.4.1. Syntax

aa)* choice clause: SOME CHOICE clause.

ab) SOME CHOICE clause: MATCH CHOICE start token, SOME MATCH CHOICE chooser clause, MATCH CHOICE finish token.

ac) SOME MATCH CHOICE chooser clause: UNITY CHOICE, SOME MATCH UNETY CHOICE alternate clause.

ba) condition: meek boolean serial clause.

bb) case: meek integral serial clause.

bc) UNITED conformity: meek UNITED serial clause.

c) SORT MOID MATCH UNETY CHOICE alternate clause: SORT MOID MATCH UNETY CHOICE in clause, strong MOID MATCH CHOICE out clause option; strong MOID MATCH UNETY CHOICE in clause, SORT MOID MATCH CHOICE out clause.

ea) SOME MATCH condition in clause: MATCH condition in token, SOME serial clause.

eb) SOME MATCH case in clause: MATCH case in token, SOME balance.

ec) SOME MATCH UNITED conformity in clause: MATCH conformity in token, SOME UNITED conformity unit; MATCH conformity in token, SOME UNITED conformity balance.

ed) SOME UNITED conformity unit: united to UNITED specification, SOME unit.

ef) meek MODE specification: open token, formal MODE declarer, MODE mode identifier option, close token, alternate token.

eg) SOME MATCH CHOICE out clause: MATCH CHOICE out token, SOME serial clause; MATCH CHOICE again token, SOME MATCH CHOICE chooser clause.

7.1. Declarers

7.1.1. Syntax

e) VICTAL structured with FIELDS declarator: structure token, VICTAL FIELDS portrayer pack.

g)* field portrayer : VICTAL FIELD portrayer.

h) deleted.

ha) VICTAL reference to MODE FOLDS portrayer: virtual reference to MODE declarer, VICTAL reference to MODE FOLDS HOMETY continuation.

hb) VICTAL NONREF FOLDS portrayer: VICTAL NONREF declarer, VICTAL NONREF FOLDS HOMETY continuation.

hc) VICTAL MODE field TAG and MODE FOLDS homogeneous continuation: MODE field TAG selector, comma token, VICTAL MODE FOLDS HOMETY continuation.

hd) VICTAL MODE field TAG HOMETY continuation: mode field TAG SELECTOR.

he) VICTAL MODE1 field TAG and MODE2 FOLDS continuation: MODE1 field TAG selector, comma token, VICTAL MODE2 FOLDS portrayer.

m) formal reference to reference to MODE declarator: reference to token, virtual reference to MODE declarer.

n) formal reference to NONREF declarer: reference to token, formal NONREF declarer.
o) VICTAL ROWS of reference to MODE declarator: VICTAL fleither option, VICTAL ROWS rower BRACKET, virtual reference to MODE declarer.
p) VICTAL ROWS of NONREF declarator: VICTAL fleither option, VICTAL ROWS rower BRACKET, VICTAL NONREF declarer.
pb) formal fleither: flexible token: either token.
pc) actual fleither: flexible token.
q) VICTAL row ROWS rower: VICTAL row rower, comma token, VICTAL ROWS rower.
ra) actual row rower: lower bound, up to token, upper bound.
rb) VIRMAL row rower: up to token option.
s) deleted.
t) LOWPER bound: meek integral unit.
u) deleted.
v) deleted.
x) deleted.
z) virtual void declarer: void token.
aa) deleted.
dd) LMOODSETY LMOOD open BOX: LMOODSETY closed LMOOD end BOX.
ee) LMOODSETY1 closed LMOODSETY2 LMOOD end BOX: LMOODSETY1 closed LMOODSETY2 LMOOD LMOOD end BOX; LMOODSETY1 open LMOODSETY2 LMOOD BOX.
ff) LMOODSETY1 closed LMOODSETY2 LMOOD1 end LMOOD2 BOX: LMOODSETY1 closed LMOODSETY2 LMOOD2 LMOOD1 end BOX.

7.4. Identifier Declarations

7.4.1. Syntax

a) identifier declaration: identity declaration; variable declaration.
b) identity declaration: MODE identity declaration; procedure identity declaration.
c) MODE identity declaration: formal MODE declarer, MODE identity definition list.
d) MODE identity definition: MODE mode identifier, equals symbol, strong MODE unit.
e) procedure identity declaration: procedure token, PROCEDURE mode identifier, equals symbol, PROCEDURE routine text.
f) variable declaration: MODE variable declaration; procedure variable declaration.
g) MODE variable declaration: heap token option, actual MODE declarer, MODE variable definition list.
h) MODE variable definition: reference to MODE mode identifier, MODE initialization option.
i) MODE initialization: becomes token, MODE source.
j) procedure variable declaration: heap token option, procedure token, reference to PROCEDURE mode

identifier, becomes token, PROCEDURE routine text.

7.5. Operation Declarations

7.5.1. Syntax

a) operation declaration: operation token, operator definition.
b) operator definition: PRAM ADIC operator, equals symbol, PRAM routine text; virtual PRAM plan, PRAM PRIETY ADIC operator, equals symbol, strong PRAM unit.

8.1. Unitary Clauses

8.1.1. Syntax

a) SOME unitary clause: SOME loop; SOME routine text; SOME assignation; SOME identity relation; SOME tertiary.
b) SOME tertiary: SOME cast; SOME PRIETY ADIC formula; SOME secondary.
c) SOME secondary: SOME generator; SOME selection; SOME primary.
d) SOME primary: SOME denotation; SOME mode identifier; SOME slice; SOME call; SOME hip; SOME ENCLOSED clause.

8.2. Coercends

8.2.0.1. Syntax

a)* coercend: COERCEND.
b)* STRONG coercend: STRONG to COERCEND.
c) deleted.
d) strong COERCEND: STRONG to COERCEND.
e) firm COERCEND: FIRM to COERCEND.
f) deleted.
fa) meek COERCEND: MEEK to COERCEND.
fb) weak reference to MODE FORM: meek reference to MODE FORM.
fc) weak NONREF FORM: unchanged from NONREF FORM; deprocedured to NONREF FORM.
G) soft reference to MODE FORM: unchanged to reference to MODE FORM; only deprocedured to reference to MODE FORM.
h) unchange to MODE FORM: MODE FORM.

8.2.1. Dereferenced Coercends

8.2.1.1. Syntax

a) dereferenced to MODE FORM: meek reference to MODE FORM.

8.2.2. Deprocedured Coercends

8.2.2.1. Syntax

aa) deprocedured to MODE FORM: meek procedure MODE FORM.
ab) deprocedured to void MORF: meek procedure void MORF.
c) only deprocedured to MODE FORM: unchanged to procedure MODE FORM; only deprocedured to procedure MODE FORM.

8.2.3. all deleted.

8.2.4. United Coercends

8.2.4.1. Syntax

a) united to union of LMOODS MOOD mode FORMSPEC: one out of LMOODS MOOD mode FORMSPEC; some of LMOODS MOOD and but not FORMSPEC.
b) one out of LMOODSETY MOOD RMOODSETY mode FORMSPEC: meek MOOD FORMSPEC.
c) some of LMOODSETY1 MOOD and RMOODSETY but not LMOODSETY2 FORMSPEC: some of LMOODSETY1 and MOOD RMOODSETY but not LMOODSETY2 FORMSPEC; some of LMOODSETY1 RMOODSETY but not MOOD and LMOODSETY2 FORMSPEC.
d) some of EMPTY and LMOOD MOOD RMOODSETY but not LMOOD2 LMOODSETY2 FORMSPEC: meek union of LMOOD MOOD RMOODSETY mode FORMSPEC.

8.2.5. Widened Coercends

8.2.5.1 syntax

a) widened to LONGSETY real FORM: meek LONGSETY integral FORM.
b) widened to structure with LONGSETY real field letter r letter e and LONGSETY real field letter i letter m FORM: meek LONGSETY real FORM; widened to LONGSETY real FORM.
c) widened to row of boolean FORM: meek BITS FORM.
d) widened to row of character FORM: meek BYTES FORM.

8.2.6. Rowed Coercends

8.2.6.1. Syntax
a) rowed to REFETY row of MODE FORM: strong REFETY MODE FORM.
b) deleted.

8.2.7. Hips

8.2.7.1. Syntax

a) strong MOID hip: MOID skip; MOID jump; MOID nihil; MOID vacuum.
e) ROWS of MODE vacuum: vacuum token.

8.2.8. Voided Coercends

8.2.8.1. Syntax

a) voided to void COMORF: unchanged from MODE COMORF.
b) voided to void MORF: deprocedured to NONPROC MORF; unchanged from NONPROC MORF.

8.3A. Loops

8.3A.1. Syntax

a) strong void loop: for part option, from part option, by part option, to part option, while part option, do part.
b) for part: for token, integral mode identifier.
c) PROBYT part: FROBYT token, meek integral unit.
d) while part: while token, meek boolean serial clause.
db) do part: do token, strong void unit.

8.3.0.1. deleted.

8.3.2. Conformity Relations

8.3.2.1. Syntax

a) boolean conformity relation: united to UNITED sample, conforms to and becomes token, meek UNITED tertiary; united to UNITED declarand, conforms to token, meek UNITED tertiary.
b) meek MODE sample: soft reference to MODE tertiary.
c) meek MODE declarand: virtual MODE declarer.

8.4. Formulas

8.4.1. Syntax

a)* formula: SOME PRIETY ADIC formula.
B) MOID PRIORITY dyadic formula: MODE1 PRIORITY operand, procedure with MODE1 parameter and MODE2 parameter MOID PRIORITY dyadic operator, MODE2 PRIORITY plus one operand.
c)* operand: MODE PRIETY operand.
d) MODE PRIORITY operand: firm MODE PRIORITY dyadic formula; MODE PRIORITY plus one operand.
e) MODE priority NINE plus one operand: firm MODE monadic formula; firm MODE secondary.
g) MOID monadic formula: procedure with MODE parameter MOID monadic operator, MODE priority NINE plus one

operand.
h)* dyadic formula: MOID PRIORITY dyadic formula.

8.6.1. Slices

8.6.1.1. Syntax

aa) REFETY ROWS of MODE slice: weak REFETY ROWSETY ROWS of MODE primary, ROWSETY ROWS leaving ROWS indexer BRACKET; weak REFETY ROWS2 of ROWS of MODE primary, ROWS2 leaving EMPTY indexer BRACKET.
ab) REFETY NONROW slice: weak REFETY ROWS2 of NONROW primary, ROWS2 leaving EMPTY indexer BRACKET.
b) row ROWS leaving row ROWSETY indexer: trimmer, comma token, ROWS leaving ROWSETY indexer; subscript, comma token, ROWS leaving row ROWSETY indexer.
c) row ROWS leaving EMPTY indexer: subscript, comma token, ROWS leaving EMPTY indexer.
d) row leaving row indexer: trimmer option.
e) row leaving EMPTY indexer: subscript.
f) trimmer: lower bound option, up to token, upper bound option, new lower bound part option; new lower bound part.
g) new lower bound part: at token, new lower bound.
h) new lower bound: meek interal unit.
i) subscript: meek integral unit.
j)* trimscript: trimmer; subscript.
k)* indexer: ROWS leaving ROWSETY indexer.
l)* boundscript: LOWPER bound; new lower bound; subscript.

8.6.2. Calls

8.6.2.1. Syntax

a) MOID call: meek procedure with PARAMETERS MOID primary, actual PARAMETERS pack.
b) actual MODE parameter: strong MODE unit.