MODE CHECKING IN ALGOL 68

by

John Donald Thomson

B.Sc., University of British Columbia, 1971

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
September, 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 **Aug 20, 1973**
Abstract

The programming language ALGOL 68 offers great flexibility in the use of modes and operators. The convenience for the programmer results in problems for the implementor in coercion, balancing, and identification of operators. This thesis presents a solution to these problems that is being successfully used in an ALGOL 68 compiler.
# Table of Contents

1. Introduction ................................................. 1  
   1.1. Special Features of ALGOL 68 .......................... 1  
   1.2. Some Implementation Terms ............................ 2  
   1.3. Description Methods ..................................... 4  
2. Program Tree Representation ..................................... 6  
   2.1. Terminology ............................................ 6  
   2.2. Nodes in the Program Tree ............................. 7  
      2.2.1. Declarer Nodes (Modes) ........................... 8  
      2.2.2. Coercend Nodes .................................. 11  
      2.2.3. Structural Nodes .................................. 13  
      2.2.4. Format Nodes ..................................... 13  
      2.2.5. Declaration Nodes .................................. 13  
3. Building the Program Tree ..................................... 15  
   3.1. Action on the Semantics Stack ......................... 15  
   3.2. The Building Routines .................................. 17  
   3.3. Tree Pruning During the Parse ......................... 21  
4. Touring the Program Tree ...................................... 23  
   4.1. Procedure Structure .................................... 23  
   4.2. Tour .................................................. 25  
   4.3. Displays (Collateral Clauses) ......................... 27  
   4.4. Slices ................................................ 30  
   4.5. Identification of Identifiers ........................ 32  
   4.6. Identification of Selectors ........................... 34  
   4.7. Identification of Calls and Slices .................... 35  
   4.8. Vicetal Checking ...................................... 36
List of Tables

Table 5.1. h Function ........................................ 44
Table 5.2. Balancing ........................................ 52
Table 6.1. Coercion ........................................ 58
Table 6.2. Coercion Contexts ............................... 59
List of Figures

Figure 2.1 .................................................... 7
Figure 2.2 .................................................... 8
Figure 2.3 .................................................... 10
Figure 2.4 .................................................... 11
Figure 2.5 .................................................... 12
Figure 2.6 .................................................... 13
Figure 2.7 .................................................... 14
Figure 3.1 .................................................... 15
Figure 3.2 .................................................... 18
Figure 3.3 .................................................... 19
Figure 4.1 .................................................... 24
Figure 6.1 .................................................... 57
Figure 6.2 .................................................... 64
Figure 6.3 .................................................... 65
Figure 6.4 .................................................... 66
Acknowledgement

My supervisor, Professor John Peck, has always been available for questions and has provided a great amount of encouragement to me. He, and the other members of the ALGOL 68 project at U.B.C. (especially Vince Manis, Dave Ramer, and Peter van den Bosch) have given me many useful suggestions and participated in many long, revealing discussions.

I am grateful for the financial support of NRC for this project.
1.1. Special Features of ALGOL 68

The most noticeable features of ALGOL 68 are its generalization of data types, modes, and the flexibility allowed the programmer to create and manipulate these modes. A programmer may construct any number of modes in a program, and may do so in a variety of ways. Either of the two constructions

\[ \text{struct (int i, j) x} \] and \[ \text{mode pair = struct (int i, int j); pair} \]

may be chosen by the programmer but are required to produce identical objects. These objects may conveniently be manipulated in formulas, since ALGOL 68 allows a programmer to define his own operators on modes he has created. For example, the declaration

\[ \text{op - = ( pair a) pair: (-i of a, -j of b);} \]

Would define an operator \( \text{op} \) acting on a data object of mode \( \text{pair} \).

As a direct result of the great flexibility of ALGOL 68, the process of mode checking (generalized type checking) is
complex. Three major mode checking algorithms are: coercion (changing one mode to another), balancing (finding a "representative" mode from a group of modes), and operator identification. This thesis describes these algorithms as they are used in a compiler, and shows how they can be efficiently executed, remain logically independent, and produce meaningful error messages.

1.2. Some Implementation Terms

The Report on the Algorithmic Language ALGOL 68 [2] states that "The syntax of ALGOL 68 is such that the parsing of a program can be performed independently of the modes of its constituents". This statement suggests a method of determining whether or not a sequence of symbols is an ALGOL 68 program. A "mode independent parse" of the symbols is first made to see if they can be generated by a "context free grammar". Then, the various constructions are examined in a "mode dependent parse" to see that the modes of the constructions meet with the syntactic requirements of the Report. This is the method used in the Vancouver implementation. The mode independent parse of a program produces a syntactic parse tree of the program, called the program tree, that will be the representation of the program for all later passes of the compiler.

In addition, the mode independent parse of the program builds a "mode table" that contains representatives of all modes
in the program. This mode table is the primary data structure of the coercion, balancing, and operator identification algorithms. The description of the coercion model of Peck [5] states that

"the mode table may be minimized so that each mode occurs only once. The determination of coercion can then be accomplished."

What this means is that if the two declayers `struct (int i, j)` and `struct (int i, int j)` appear in the program and are initially put into the mode table, one "copy" may be discarded from the mode table when they are discovered to be equivalent. The coercion algorithm model makes use of the fact that there are not two copies of an equivalent mode in the mode table.

While this method of dealing with equivalent modes is adequate for a model of coercion, it is not quite adequate in a compiler because the mode representatives for row and rowof modes must contain bound information. The two declayers `struct ([1:5]int i, bool j)` and `struct ([1:10]int i, bool j)`, for example, are of the same mode. One copy or mode representative cannot be thrown away since the bound information may be different, and must be preserved until code generation time.

To conform to standard practice, this document still speaks of a "mode table", although it may more accurately be thought of as a "declarer table", consisting of representatives of all declayers in the program. One piece of information associated with each declarer representation is the equivalence class or
mode of which that declarer is a member. Thus in the example above, the two declarers would be kept separately and have different bound information, but would belong to the same class (be of the same mode).

1.3. Description Methods

Numerous references to the Report and the Revised Report appear throughout this document. These are, respectively, to the Report on the Algorithmic Language ALGOL 68 [2] and to the Draft Revised Report on the Algorithmic Language ALGOL 68. A section of the Revised Report, where appropriate, is indicated within the braces "{" and ""}, for example {6.1.d}. Since the latter document is still changing somewhat, sections may be slightly different in the final version.

PL360 is the language that is used to implement ALGOL 68 in Vancouver. The size of programs in this language makes it impractical to include a source listing of the mode checking algorithms, even as an appendix. To include algorithms in the "step 1, ... , step 9a" format would be self defeating, since this thesis deals with ALGOL 68, an "algorithmic language" "serving many people in many countries" [2]. For these reasons, important algorithms have been included as ALGOL 68 procedure declarations, with long, "meaningful" identifiers and selectors. It is hoped that algorithms presented in this fashion are more readable.
For brevity, common abbreviations such as "ref" for "reference to" and "L int" for "LONGSETY integral" are used in many situations where the meaning is clear.
Chapter 2.

Program Tree Representation

2.1. Terminology

As the mode independent parse of an ALGOL 68 program proceeds, a record must be kept of the syntactic constructions which occur. This information is needed for later mode checking and code generation passes of the compiler. The implementation at Vancouver uses the syntactic tree of the parse as the basic form of this record and calls it the program tree.

The program tree should be thought of as the essential substance of the syntactic parse tree, i.e., those parts which have no semantic content and are necessary only for parsing have been eliminated. In addition, space is left in the program tree at those places where information must later be placed. Some simple tree pruning and compaction is done as the program tree is built.

For ease of discussion, the nodes of a program tree and the sons of that node are given names associated with the production rule involved in that node. For example, an "assignation" node
with a "tertiary" son and a "unit" son is suggested by the context free production rule:

assignation: tertiary, becomes symbol, unit.

(here the becomes symbol is discarded because it no longer possesses semantic value).

2.2. Nodes in the Program Tree

The structure of the program tree nodes is given in the figures that follow. The program tree nodes are divided into five classifications, according to their uses: declarer nodes, coercend nodes, structure nodes, format nodes and declaration information nodes. Within each of these classifications, there are common fields (the head). For example the first five fields in the assignation node (a coercend) are common to all coercend nodes:

```
<table>
<thead>
<tr>
<th>type</th>
<th>nf=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>→ → coercion sequence</td>
</tr>
<tr>
<td></td>
<td>→ → &quot;tertiary&quot;</td>
</tr>
<tr>
<td></td>
<td>→ → &quot;unit&quot;</td>
</tr>
</tbody>
</table>
```

Figure 2.1
The last two fields (the tail) are unique to the assignation node.

2.2.1. Declarer Nodes

The declarer nodes contain all the information needed for the elaboration of a declarer. They are divided into equivalence classes (modes) that are used in the mode checking pass of the compiler. The "reference to mode" node below shows the structure of the head for all declarer nodes:

```
<table>
<thead>
<tr>
<th>type</th>
<th>nf=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>mn</td>
</tr>
<tr>
<td>cw</td>
<td>h</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
</tbody>
</table>
```

- **type** = type of node
- **nf** = number of subfields
- **dn** = declarer number
- **mn** = mode number
- **cw** = coercend word
- **h** = the balancing function

Figure 2.2

The function h is used in balancing. The flags and link fields are used in various algorithms to flag the declarers and to link them together in a list. The coercion word is used during
coercion to hold an item of a coercion sequence. When
equivalencing is done, the mode number is filled with the
equivalence class to which the declarer belongs. Thus cw, flags
and link are essentially scratchpads.

It is necessary to keep copies of two declarers of the same
mode because their bounds (or bounds of their subdeclarers) may
be different. For example, the following structured declarers
must be kept separate although they specify the same mode:

\[
\text{struct } ([1:5] \text{int } x, \text{ real } y) \\
\text{struct } ([1:10] \text{int } x, \text{ real } y)
\]

In an ALGOL 68 program it is possible to use a declarer
before it is defined, for example:

```algebra
mode a = struct (b x, y);
mode b = ref a;
```

Because of this, temporary declarers are created during the
parse of a program. They are eliminated after the parse is
completed.

Slices and selections cause another complication in ALGOL
68. In this example, the mode of the selection and the slice is
reference to integral:

```algebra
[1:2] \text{int } a;
\text{struct (int } i, j) x;
a[1] := i \text{ of } x := 1;
```

Notice that starting with the modes of a and x, namely reference
to rowof integral and reference to structured with integral
letter i integral letter j, there is no simple path in the mode
tree to the mode reference to integral. One must start with the mode integral, and search through all modes until reference to integral is found. To avoid this search being done each time a slice or selection is used in the program, there is an extra field in the row, rowof and field declarer nodes where the reference to mode information is kept. For the above example the declarer nodes for rows would be built:

```
<table>
<thead>
<tr>
<th>rowof</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>mn</td>
</tr>
<tr>
<td>cw</td>
<td>h</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>pair</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #coor #</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>--</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>ref</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>mn</td>
</tr>
<tr>
<td>cw</td>
<td>h</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>prim.</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>mn</td>
</tr>
<tr>
<td>cw</td>
<td>h</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
<tr>
<td>integral</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 2.3
2.2.2. Coercend Nodes

At certain places in the syntax of ALGOL 68, coercions are performed. This involves taking a value of some mode and changing it into a corresponding value of another mode. For example, in the assignation `loc real:= 1`, `1` must be coerced from an integral value to a real value. This coercion, called widening from integral to real, must be placed somewhere in the program tree during mode checking so that it may be found at the appropriate time during code generation. Coercend nodes are those nodes whose associated values may be coerced, for example:

\[
\begin{array}{c|c}
\text{deno.} & \text{nf=3} \\
\hline
\text{range} & \text{coor} \\
\hline
\text{---\rightarrow coercion sequence} \\
\hline
\text{---\rightarrow denotation mode} \\
\hline
\text{deno acc} \\
\end{array}
\]

Figure 2.4

A pointer to a coercion sequence is part of the head of coercion nodes, after type, range and coordinate information.

Two coercend nodes that deserve special attention are the routine node and the "calice" node (i.e., a mode which may be a call or slice). The mode of a routine must be constructed and included as part of the routine node for use by the mode checking routines. In the program:
begin
  ((int i) int : i+1)(1)
end

The mode procedure with integral parameter yielding integral must be included in the routine node, in the place indicated by routine mode below:

```
<table>
<thead>
<tr>
<th>routine</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|         |   | --- > coercion sequence
|         |   | --- > formal p.p. or nil
|         |   | --- > "unit" mode
|         |   | --- > "unit"
|         |   | --- > routine mode
```

Figure 2.5

It is impossible to tell at parse time, in the Vancouver implementation, whether a(i,j) is a call or slice. For this reason a "calice" node is created at parse time. During the mode checking pass these will be resolved. The call, slice and calice nodes are identically structured, so that this may be done by a simple change in the type field.
2.2.3. Structural Nodes

These program tree nodes have no special properties; they merely preserve the structure of the program for later passes. For example, a unit series is stored in the following way:

```
[ unit ser. ] nf=n
[ range # ] coor #
   ------------------------->unit 1
   ...
   ------------------------->unit n
```

Figure 2.6

2.2.4. Format Nodes

Because of the complexity of ALGOL 68 formats, it is necessary to construct format nodes to preserve the structure of the parse. The major reason formats cannot be broken down into linear strings at this time is dynamic replicators - replication factors that are elaborated at run time.

2.2.5. Declaration Nodes

Declaration nodes are built each time an identifier, indication or operator, is defined in the program. The head of
these declaration nodes contains type and range information as in other nodes. As well there is a cross-referencing field and a field used to link all definitions of the same lexical symbol together. This linking is needed to identify an applied occurrence. There are four of these declaration nodes for modes, identifiers, operators (parameters and body), and priorities (of some operator). For example, the following node is created for each mode declaration in the program:

```
| mode dec | 4 |
| range # | coor # |
| • ----> indlink |
| • ----> xreflink |
| mode dec # |
| • ----> mode declarer |
```

Figure 2.7
3.1. Action on the Semantics Stack

The various parts of a program tree node are collected on a semantics stack during the parse of an ALGOL 68 program. When all the parts have been collected and the reduction involving an associated production rule occurs, the node is built, the parts on the semantic stack are removed, and a pointer to the node is placed on the semantics stack. During a read operation of the parser (see [10]), any terminal symbol which is read and has some semantic content (a semantic terminal) is placed on the semantics stack. It is then considered part of some node. For example, we will trace through the semantics stack during the parsing of the assignation: \( x := 1 \).

1. Action of parser: read (tag symbol)

Result in semantics stack:
2. Action of parser: reduce (primary: tag symbol)

Result in semantics stack:

```
<p>| | |
|   |   |</p>
<table>
<thead>
<tr>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 1</td>
</tr>
</tbody>
</table>
```

3. Action of parser: read (becomes symbol)

Result in semantics stack:

unchanged.

4. Action of parser: read (integral denotation symbol)

Result in semantics stack:

```
<p>| | |
|   |   |</p>
<table>
<thead>
<tr>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 1</td>
</tr>
</tbody>
</table>
```
5. Action of parser: reduce (denotation: integral denotation symbol)

Result in semantics stack:

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-----</td>
</tr>
</tbody>
</table>

• ———> denotation node
• ———> tag node
```

6. A number of reductions not affecting the semantics stack that parse the primary as a tertiary, and the known denotation as a unit.

7. Action of parser: reduce (assignation: tertiary, becomes, unit)

Result in semantics stack:

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

• ———> assignation node
```

Figure 3.1

3.2. The Building Routines

The changes in the semantics stack that result in the program tree nodes being built are accomplished by calling certain node building routines. These are designed to handle
the two different types of nodes: fixed length (e.g., assignation) and variable length (e.g., unit series).

Figure 3.2

The inclusion of variable length nodes slightly changes the program tree structure from that of the syntactic parse tree, e.g., the unit series node of the syntactic parse tree might look like:
Figure 3.3

Compacting various lists, series, trains, etc. into one node results in a saving of storage, and speeds up later passes through the program tree. When a node of variable length is being built on the semantics stack, a counter of its current length must be kept there and updated when necessary. The following routines are used to manage the semantics stack:

- **buildblock**(type, mask)**:
  
  this routine is called to build a node of type `*type*` from items on the semantics stack. These items are removed and replaced by a pointer to the node just built. The `*mask*` is used to distinguish between different forms of the node (e.g., between "loop: for part, do clause" and "loop: do clause").

- **increment**

  this routine is called to increment by one the count on the top of the semantics stack which is the length of the top variable length node, e.g., `*increment*` would be called for the production rule

  unit series: unit series, go on symbol, unit.

- **startblock**(type, num)**:

  start a variable length node of type `*type*` with the top `*num*` items on the semantics stack. Initialize the length
count to \texttt{num}. For example, \texttt{startblock(ustype,1)} would be called for the production rule

unit series: unit.

\texttt{endblock(type, depth)}

build a variable length node of type \texttt{type} which is \texttt{depth} items from the top of the semantics stack. The items making up this node on the semantics stack are removed and replaced by a pointer to the node. For example, \texttt{endblock(ultype,0)} would be called for the production rule

unit list proper pack: open symbol, unit list proper, close symbol.

The following ALGOL 68 procedure declarations give algorithms for \texttt{buildblock}, \texttt{startblock}, \texttt{increment} and \texttt{endblock}, assuming the given declarations:

\texttt{mode ptr=union(int,ref node);}  
\texttt{int top;}  
\texttt{[1:100] ptr stack;}  

\texttt{proc startblock=(int type, num)void:}  
\texttt{stack[top+:=1]:=num;}  

\texttt{proc increment=void:}  
\texttt{begin}  
\texttt{ptr i:=stack[top];}  
\texttt{int j:=case stack[top-1] in (int k):k esac+1;}  
\texttt{stack[top]:=j; stack[top-1]:=i}  
\texttt{end;}
3.3. Tree Pruning During the Parse

In addition to ignoring terminals with no semantic content, there are other methods used to make the program tree smaller than the syntactic parse tree without losing essential information.

Many production rules (e.g., secondary: primary) are not used to group parts of a construction together, but are necessary for parsing purposes only. A "secondary" node is useless for further passes and is therefore never built. Eliminating these nodes results in significant savings since 25%
of the rules are of this type.

Another easy way to reduce the size of the program is to eliminate nodes such as unit series, if there is only one member of the series. This may save quite a lot of space for some programs, since about 10% of the production rules involve lists, series, sequences, etc.
Chapter 4.

Touring the Program Tree

4.1. Procedure Structure

The mode checking pass of the Vancouver ALGOL 68 compiler consists of a main routine tour that visits all of the nodes of a program tree and, at appropriate places, calls the identification, coercion, and balancing routines. tour is a recursive routine whose purpose is to set up parameters to the mode checking routines in such a way that these routines need know nothing of the structure of the program tree.

The following figure illustrates the hierarchy of the various routines in the mode checking pass (a routine "points" to all routines that it calls):
The following sequence of calls would be made by checking the assignation in `int i; i:=1;`:

```
tour(strong,assignation,void)
tour(soft,tag,nil)
  identify identifier(i)
  coerce(soft, #from# ref int, #to# nil)
tour(strong,denotation,int)
  coerce,strong,#from# int,#to# int)
coerce,strong,#from# ref int,#to# void)
```
A compiler pragmatic will produce traces of this form (see [11]).

The call \texttt{tour(strong, root of program tree, void)} initiates the mode checking pass. As in other cases, the connection between the syntax of the Revised Report and the calls of \texttt{tour} can be clearly seen:

\{2.2.1\} program: strong void new closed clause.

4.2. Tour

The procedure \texttt{tour} must decide when to call identification, coercion, and balancing routines. Deciding when to call the identification routines is an easy task (e.g. \texttt{identify selection} is called when at a selection node in the program tree). Calling the coercion and balancing routines is a little trickier. In order to make the balancing routine independent of the structure of the program tree, all modes of the balance must be available when \texttt{balance} is called. To accomplish this, it is necessary to "tour" the tree in the states: strong, firm, meek, weak, soft, and error corresponding to the SORTs of coercion (plus a state of error). Deciding when to call \texttt{balance} and \texttt{coerce} is based on these states. In a strong position, the aposteriori mode is also passed to \texttt{tour}.

The parameter lists for \texttt{balance} and \texttt{coerce} are built up
on pair stack. The items of this stack consist of a mode field, and a field which is a pointer to where the coercion sequence associated with that mode may be stored. A count of the number of "pairs" on the stack is always on the top of the stack.

There are two interesting types of nodes as far as coercion is concerned: "coercends", the objects which may be coerced, and "coercers", the objects which syntactically initiate a coercion or balance e.g., integral enquiry clause which is a meek integral series. If a coercend is arrived at in a strong position (the aposteriori mode will be known) then the procedure •coerce• is called directly. Otherwise, the aposteriori mode will not, in general, be known and the mode of the coercend and its coercion sequence will be placed on the pair stack. The actions at the assignation node and the denotation node are given below in an attempt to clarify this explanation.

```plaintext
proc tour=(int sort, ref node n, md apost)void:
begin
    ref md apriori, newapost;
    case node type of n in
        # denotation #
        # a denotation is a coercible object #
        at a coercend(sort, mode of n, cs of n),
            # assignation #
            # an assignation initiates coercion as well as being itself a coercible object #
            pairstack[top+:1]:= (0, nil); # set count to zero #
            tour(soft, asstertiary of n, skip); # stack lhs modes #
        if
            at a coercer(soft, apriori, newapost)
                # soft coercion returns apriori and
                new aposteriori modes #
        then
```
clear(pair stack);
tour(strong,assunit of n, newapost);
at a coercend(sort,apriori,cs of n)
else
  error("no possible soft balance");
clear(pair stack);
tour(error,assunit of n,newapost)
fi
esac
end

Here the procedure «at a coercend» decides whether to call «coerce» immediately, or stack a mode to be coerced later. The procedure «at a coercer» decides whether to balance or coerce:

proc at a coercend=(int sort,md m,ref cs c)void:
  if sort=strong then
    coerce(strong,m,apost,c)
  elsif sort=error then
    int i:=m of pair stack[top]; i+:=1;
    pair stack[top]:=(m,c);
    pair stack[top+:=1]:= (i,nil)
  fi

proc at a coercer=(int sort,type ref md apriori, newapost)bool:
  begin
    int i:=m of pair stack[top];
    [1:i] pair a:=pair stack[top-i-1:top-1];
    if i>1 then
      balance(sort,type,i,a)
    else
      coerce(sort,m of a[1],newapost,c of a[1])
    fi
  end

4.3. Displays (Collateral Clauses)

The change U33 of [7] in the language ALGOL 68 greatly eases many implementation problems. This change, restricting collateral clauses to a strong position, resulted from
ambiguities in operator identification discovered by Wüssner and Yoneda [1]. The following section of an ALGOL 68 program is the example given by Yoneda:

```algon68
mode a = [1:1] s;
mode c = [1:2,1:1] s;
mode s = struct(ref a f,g);
a a; s s;
op ? = (c x) int: skip ;
op ? = (s x) int: skip;
? if bool then s else (a,a) fi
```

If the display *a(a,a)* is allowed in a firm position, then the identification of *a?* depends on whether the display is interpreted as a row display or a structure display. If *a(a,a)* is taken as a structure display of mode **s** then the second declaration of *a?* is identified. But it may also be taken as a row display of mode **c** in which case the first declaration of *a?* is identified.

The restriction that displays must be strong simplifies weak and firm balancing, since all modes that could possibly be in the weak or firm position of the balance are now known. If encountered as part of a balance, displays may be set aside until the mode of the balance is found. Then the displays can be simply handled since their mode is now known.

The following syntax from the Revised Report [3.3.1] indicates clearly how displays must be handled:

strong void NEST collateral clause:
strong void NEST joined portrait PACK.
strong void NEST parallel clause:
parallel token, strong void NEST collateral clause.
strong ROWS of MODE NEST collateral clause:
  where <ROWS> is <row>,
  strong MODE NEST joined portrait PACK;
where <ROWS> is <row ROWS1>,
  strong ROWS1 of MODE NEST joined portrait PACK.
strong structured with FIELDS FIELD mode NEST collateral clause:
  NEST FIELDS FIELD portrait PACK.
NEST FIELDS FIELD portrait:
  NEST FIELDS portrait, comma token,
  NEST FIELD portrait.
NEST MODE field TAG portrait:
  strong MODE NEST unit.
* structure display: strong structured with
  FIELDS FIELD mode NEST collateral clause.
* row display: strong ROWS of MOST collateral clause.

All displays must be either void, ROWS of MODE, or
structured with FIELDS mode. If void, then all constituents of
the display must be examined and strongly coerced to void. If
row ROWS of MODE (or row of MODE), then the constituents of the
display must be strongly coerced to ROWS of MODE (or MODE). If
structured with FIELDS, then each constituent must be strongly
coerced to the mode of its associated FIELD. The following
algorithm is used in the Vancouver implementation to accomplish
this. This algorithm is called when the aposteriori mode of the
display is discovered.

proc display = (ref node n, md apost) void:
  if apost=void then
    for i to number of subtrees of n do
      begin
        node t:= (subtree of n)[i];
        tour (strong,t,void)
      end
  elseif modetype of apost=structure then
    int i=number of subtrees of n;
    if i=number of subfields of apost then
      error("incorrect length of structure display")
    else
      for j to i do
        begin...
node \( t \) := (subtree of \( n \))[\( j \)];
md \( m \) := field mode of (sub field of apost)[\( j \)];
tour (strong, \( t \), \( m \))
end
fi
else
modetype of apost=row or modetype of apost=rowof then
md \( m \) := submode of apost;
for \( i \) to number of subtrees of \( n \) do
begin
node \( t \) := (subtree of \( n \))[\( i \)];
tour (strong, \( t \), \( m \))
end
else
error ("display must be void, row or structure")
fi;

4.4. Slices

The mode of a slice in ALGOL 68 depends upon the mode of its primary, and the number of trimmers in the slice. There are really two types of slicing, one done on values of mode ROWS of MODE, and one done on values of mode reference to ROWS of MODE. The following examples illustrate this:

<table>
<thead>
<tr>
<th>mode of a slice</th>
<th># of trimmers</th>
<th>mode of slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>([,] ) int [a[1,2]]</td>
<td>0</td>
<td>int</td>
</tr>
<tr>
<td>([,] ) int [a[1:2,3]]</td>
<td>1</td>
<td>([,] ) int</td>
</tr>
<tr>
<td>([,] ) int [a[1:2,3:4]]</td>
<td>2</td>
<td>([,] ) int</td>
</tr>
<tr>
<td>ref[(,)] int [a[1,2]]</td>
<td>0</td>
<td>ref int</td>
</tr>
<tr>
<td>ref[(,)] int [a[1:2,3]]</td>
<td>1</td>
<td>ref[(,)] int</td>
</tr>
<tr>
<td>ref[(,)] int [a[1:2,3:4]]</td>
<td>2</td>
<td>ref[(,)] int</td>
</tr>
</tbody>
</table>

The precise connection between the modes and the number of
trimmers is defined in the Revised Report {5.3.2.1} as follows:

REFETY MODE1 NEST slice:
weak REFLEXETY ROWS1 of MODE1 NEST PRIMARY,
ROWS1 leaving EMPTY NEST indexer STYLE bracket,
where <REFETY> is derived from <REFLEXETY>;
where <MODE1> is <ROWS2 of MODE2>,
weak REFLEXETY ROWS1 of MODE2 NEST PRIMARY,
ROWS1 leaving ROWS2 NEST indexer STYLE bracket,
where <REFETY> is derived from <REFLEXETY>.
row ROWS leaving ROWSETY1 ROWSETY2 NEST indexer:
row leaving ROWSETY1 NEST indexer, comma token,
ROWS leaving ROWSETY2 NEST indexer.
row leaving EMPTY NEST indexer: NEST subscript.
row leaving row NEST indexer: NEST trimmer;
NEST new lower bound option.
NEST subscript: meek integral NEST unit.
NEST trimmer: NEST lower bound option,
up to token, NEST upper bound option,
NEST new lower bound option.
NEST new lower bound:
at token, NEST lower bound.

Each time a slice node is encountered during the tour of a program tree, the number of trimmers in the trimscript list is counted. This, together with the mode of the primary is used to calculate the mode of the slice. In addition, the total number of trimscripts in the slice is checked against the mode of the primary. The following algorithm is used:

```
proc at a slice=(ref node n, ref md a)void:
begin
int ts,t;
md m;
ts:=number of trimscripts(trimscripts of n);
t:=number of trimmers(trimscripts of n);

#find new mode of slice #
if mode type of a = reference then
    for i to t do
        begin
            m:=refsuob of a;
            if mode type of m=rowoftype and i<t then
```
error("too many trimscripts in slice");
go to exit
fi;
a:=refrowsub of m
end;
m:=refs of a
else
for i to t do
begin
if modetype of a=rowoftype and i<t then
  error("too many trimscripts in slice");
go to exit
fi;
a:=rowsub of a
end;
m:=a
fi;

# check number of trimscripts #
for i to ts-t do
begin
if modetype of m=rowoftype and i<t then
  error("too many trimscripts in slice");
go to exit
else if modetype of m=rowoftype and i=t then
  error("too few trimscripts in slice");
go to exit
fi;
m:=rowsub of m
end;
exit:
  skip
end

4.5. Identification of Identifiers

All mode and label identifiers must be identified during the mode checking pass through the program tree. During the mode independent parse of the program, all defining occurrences of a given identifier are placed in a linked list. The items in this list contain range information and mode information about that identifier. The list is ordered from highest range number
(youngest range) to lowest range number (oldest range) to make lookup more efficient.

As the program tree is toured, active ranges are placed on a range stack when a range is entered and are removed from this stack when the range is left. The declaration identified is the one with the highest numbered (youngest) range which is currently on the range stack. A pointer to this declaration is put in the identifier node of the program tree for use by later passes. The following algorithm is used:

```plaintext
proc identify identifiers=(ref node n) bool:
begin
  deflist d:=identifier definitions[tag of n];
  int t:=top of range stack;
  int active range:=range stack[t];
  int definition range:=range of d;
  bool can identify:=active range=definition range;
  while ~can identify
    rep
      while active range>definition range
        rep
          t:=t+1;
          if t=0 then error exit fi;
          active range:=range stack[t]
        end rep;
      end while
    end rep;
    d:=next of d;
    if d=nil then error exit fi;
    definition range:=range of d
    can identify:=active range=definition range
  end while
  tag of n:=d; goto e;
error exit:
  error("identifier cannot be identified");
end
```
4.6. Identification of Selectors

During the mode checking pass, all selections must be identified with the appropriate field of a structure. The inclusion in ALGOL 68 of multiple selections [7], makes it necessary to do some searching for a mode with row prefixes. This is done by the procedure `findmode`. The following algorithm is used to identify selections:

```plaintext
proc identify selectors= (ref node n, ref md a) bool:
begin
  bool can identify:= false;
  int num refs:= 0, num rows:= 0;
  md m;
  if mode type of a= reference then
    a:= refsub of a;
    num refs+:= 1;
  fi;
  while modetype of a= row or modetype of a= rowof rep
    a:= rowsub of a;
    num rows+:= 1
  per;
  for i to number of submodes of a while ~ can identify rep
    m:= (structsub of a)[i];
    can identify:= field tag of m= tag of n
  per;
  if can identify then
    field of n:= m;
    a:= findmode (num refs, num rows, a)
  else
    error ("selection cannot be identified");
  fi;
  can identify
end
```
4.7. Identification of Calls and Slices

Until mode checking has been done, there is no way to tell whether a construction such as \( \text{if } b \text{ then } c \text{ else goto } 1 \text{ fi}(1,2) \) is a call or slice. Therefore, the implementer must balance some modes without even knowing the sort of the balance. What is known, however, can be used to find the sort of the balance quickly. Since the construction involved must be either a call or a slice, the balance must be either meek to procedure with parameters, or weak. To test whether a meek balance to procedure with parameters is possible, attempt to coerce the first non-hip, non-display mode to procedure with parameters. If this succeeds, then the balance must be meek or there is no possible balance. If it fails, then the balance must be weak or not be possible.

```plaintext
proc identify calices=
(int num modes, [:]md m, ref int type)void:
begin
    md p:=some procp mode;
    for i to num modes do
        if
            m[i]=a hip or
            m[i]=collateral modes
        then
            if coerce(meek, m[i], p) then
                type:=call
            else
                type:=slice
            fi;
            goto f;
        fi;
    fi;
    error("call or slice cannot be identified");
f:
    skip
end;
```
4.8. Victal Checking

Declarers in ALGOL 68 are of three types: virtual, actual, and formal. Since it is inconvenient to check these types during the mode independent parse, they are checked during the mode checking pass. Formal and virtual rows may not contain bounds, and actual rows must have them. Thus the following examples are incorrect:

\[
\text{ref[1:10]int a=loc[1:10]int;}
\]
\[
\text{ref[]int a=loc[]int;}
\]

And must be written:

\[
\text{ref[]int a=loc[1:10]int;}
\]

The declarer \text{void} is restricted in use to void casts, constituents of united modes, and the modes yielded by procedures.

The victal check consists merely of examining all declarers in either of the states virtual, actual, or formal and checking that they have appropriate bounds. The states are determined by the rules of the Revised Report \cite{revised-report}.

4.9. Error Recovery

If one part of a construction is incorrect, what should be done about mode checking the rest of it? For example, in the assignation of \text{int z=1; z:=3.0}, the left hand side is examined and found to be in error (the mode of \text{z} must begin with a
ref). What then should be the action taken on the right hand side of the assignation (it also would be in error if the mode of \( z \) were \( \text{ref int} \))? Two possibilities would be:

1. Assume the mode of the right hand side is \( \text{int} \), and proceed as always.
2. Do not even consider the right hand side, since the left hand side is in error.

Neither of these methods is good, in general. The first approach, to make assumptions about the modes, is difficult in ALGOL 68 because of the large number of possible modes. It would tend to produce many error messages where no error exists and is would be difficult to separate them from the real errors. The second method does not give the programmer as much information about his program as is possible. This can be seen from the example:

```plaintext
proc int i=skip; [1:3]real y; int z=2;
i:=(skip | y[1,2] | z:=2)
```

Even though the left hand side, \( i= \), of this assignation is incorrect, there are three errors on the right hand side that would be errors no matter what the left hand side would be.

The method of dealing with this problem in the Vancouver compiler is to examine the right hand side in a special error state. While no errors are detected in this error state, it is possible to get into a known state in a part of a construction that is being considered in the error state. In the above
example, it is still possible to detect errors in \texttt{skip} (no meek coercion to \texttt{bool}), \texttt{y[1,2]} (too many trimscripts), and \texttt{z:=2} (\texttt{z} does not begin with a \texttt{ref}) even though nothing is checked of the modes of the slice, \texttt{y[1,2]}, and of the assignation, \texttt{z:=2}.

This method has the following properties:

1. No error messages are generated because of assumptions about earlier errors.

2. All "certain" errors in sub-constructions of erroneous constructions can be detected.

There is an unfortunate consequence of the method of mode checking an ALGOL 68 program. One hundred errors, all the same (\texttt{real} cannot be coerced to \texttt{int}), are found in a construction \texttt{[ ]int a=(1.0,...,100.0]}. There is clearly only one conceptual error here, and we feel justified in making a small effort to avoid printing out 100 error messages. The following algorithm in the error printing routine insures that an error that occurs numerous times separated by only one coordinate (comma or go on symbol -- see \cite{[11]}) will be printed only 3 times:

```plaintext
if
    this error=last error and
    this coor = last coor+1
then
    error count +=1
elsef
    error count>3
then
    print("previous error detected","error count","times");
```
error count:=1
else
  error count:=1
fi;
if error count\leq3 then print(this error) fi;
last error:=this error; last coor:=this coor;
5.1. Definition of Balancing

In ALGOL 68, constructions such as serial clauses and conditional clauses have a mode associated with them. This makes it possible to write constructions like \texttt{real} \texttt{x, ref real y; (p|x|y):=1.0* and int i, j; a[1:(p|i|j)]}. The mode of the clause may be derived from the constituent modes in the clause and the position of the clause in the program or may be known without examining the modes within the clause. Thus, in the first example, "reference to real" can be derived from the modes of \texttt{x} and \texttt{y}, and in the second example, "integral" is known to be the mode of the clause, whatever \texttt{i} and \texttt{j} might be. In either case, the language makes the same requirement to insure that the program "makes sense": that one of the constituent modes of a \texttt{SORT} clause be \texttt{SORTly} coercable to the a posteriori mode of the clause, and that the other modes of the clause be strongly coercable to that same mode of the clause. This requirement means that the a posteriori mode of the clause, disregarding coercions of a given \texttt{SORT}, will be the same as one of its
constituent modes. The selection of this mode from the constituents of a clause is known as balancing.

5.2. The Need for Efficient Balancing

Neither the Report nor the Revised Report gives any hint of how the SORT position of a balance can be found, other than by a process of trial and error. While this approach may be fine for the clause \((p|x|y):=1.0\)\textsuperscript{e}, in general trial and error is too expensive to use. In a case clause like the following, each time a candidate for the SORT position, \(y[i]\), is examined, \(n\) strong coercion attempts may be made before one fails (assuming the candidates are chosen in order):

\[
[1:n] \text{ref int } y; \text{ int } x;
\text{case } i \text{ in}
\begin{align*}
\text{... } y[1], \\
\text{... } y[n]
\end{align*}
\text{out}
x
\text{esac:=1.}
\]

This is because for each \(i\), only the mode of \(x\) ("reference to integral") cannot be strongly coerced to the mode of \(y[i]\) ("reference to reference to integral"). Before the correct candidate \(x\) is chosen, an embarrassing amount of time may be spent if the value of \(n\) is large.
5.3. The Development of Balancing

Helge Scheidig [3] in his doctoral thesis of 1970 presented a quick method for determining the soft and weak positions for soft and weak balances. This method involved counting the ref and row prefixes which are at the beginning of a mode. For soft balances, Scheidig showed that the constituent mode of the balance with the minimal number of ref prefixes (and maximal number of row prefixes if two modes had the same number of ref prefixes) must be the mode in the soft position of the balance. For weak balances, the mode with the maximal number of row prefixes must be the one in the weak position of the balance.

In 1972, Scheidig [6] stated (without giving a method) that the firm position of a firm balance could be found if the positions of collateral clauses in the language were restricted. He pointed out that these restrictions were also necessary to avoid ambiguity in the identification of operators.

In September, 1972, IFIP Working Group 2.1 approved extensive changes to the language ALGOL 68 [7], [8]. Among these changes were two that affected balancing: C58 (more meekness) and U33 (strong displays). The addition of more meek positions, especially the UNITED choice in the conformity clause, made it necessary to do meek balancing. The restriction of displays to a strong context cleared up operator ambiguity problems (see chapter 4) and also made it possible to do a fast firm balance.
Early in 1973, W. J. Hansen proposed a balancing function \( h \), acting on a mode and yielding an integer value, that could be used to find the firm position of a balance. Numerous modifications to this function were made at the University of British Columbia and the University of Alberta (Edmonton) to make it handle meek, weak, and soft balancing as well as firm balancing. Hendrik Boom [9] in June, 1973 put forward an improved function \( h \), acting on a mode and yielding a rational value, that could be used for any SORT of coercion. Since that time, some simplifications (due to L. Meertens) have been made in the function making it easier to describe.

The following result using the balancing function to order modes solves the problem of efficient balancing (some problems may arise if a suggested change forces balancing of transients, see the Revised Report {3.2.1.k}):

**General Balancing Theorem**

Let \( m_1, \ldots, m_n \) be \( n \) modes \((n \geq 1)\).

Let \( m \in \{m_1, \ldots, m_n\} \) be such that:

\[ h(m) \geq h(m_i) \text{ for all } i=1, \ldots, n. \]

Then in any SORT balance of objects of modes \( m_1, \ldots, m_n \) together with some hips and some displays, \( m \) may be in the SORT position of the balance.
5.4. General Balancing Theorem

5.4.1. Definition of $h$

The following definition of $h$ is one given by L. Meertens. The function $h$ has an integral part $H$ and a fractional part derived from a function $r$ that keeps track of the number of refs. This separation is necessary to insure that the number of refs, a count necessary for soft coercion, can safely be ignored for non-soft coercion.

$$h(m) = H(m) + 2^r(m)$$

$H(\text{ref ROWS of } m) = H(\text{ROWS of } m) - 1$

$H(\text{ref } m) = H(\text{proc } m) = H(m)$

$H(\text{L real}) = 2$

$H(\text{L compl}) = 4$

$H(\text{rows of } m) = 2n + H(m)$ where there are $n$ rows in ROWS

$H(\text{union (m1, ..., mn)}) = 2n + \max \{H(m1), ..., H(mn)\}$

$H(m) = 0$ in all other cases

$r(\text{ref } m) = r(m) + 1$

$r(\text{proc } m) = r(m)$

$r(m) = 0$ in all other cases

Table 5.1. h Function
5.4.2. Properties of \( h \)

The numerical values of \( h \) on a mode are unimportant provided that the following properties are preserved. They will be referenced later in the proofs of the General Balancing Theorem.

(H1) A series of coercions containing a strong or firm (but not meek) coercion, except voiding, (i.e., rowing, widening, uniting) increase the value of \( h \) by at least 1. They increase \( H \) by at least 1. Further, if the coercion is not to \texttt{ref ROWS} of \( m \), then \( h \) is increased by at least 2 (since \( H \) is).

(H2) A series of meek coercions affect the value of \( h \) by less than 2. This is because the value of \( H \) may be affected by only 1 (first rule) and the value of \( 2^{**(-r)} \) will be affected by less than 1. Further, a series of meek coercions to \texttt{ref ROWS} of \( m \) increase \( h \) by less than 1 (since the first rule for \( H \) cannot apply).

(H3) \( h \) is unaffected by soft coercion. Neither \( H \) nor \( r \) is.
(H4) All non-soft coercions to reference to mode increase the value of h. This is because r decreases (hence $2^{*(-r)}$ increases) while H is unaffected, since the first rule cannot apply.

5.4.3. Coercion Properties

The following properties of coercion are used to prove the General Balancing Theorem. The appropriate syntax of the Revised Report is given to indicate how these properties have been derived.

(C1) No strong (but not meek) coercion can occur before uniting. This is clear from the rule:

{6.4.1} united to UNITED FORM: MEEK MOID FORM, where MOID unites to UNITED.

(C2) No meek coercion can occur after uniting. This is because dereferencing and deproceduring are the meek coercions, and they must be on reference to and procedure modes respectively:

{6.2.1} dereferenced to MODE1 FORM:
MEEK REF to MODE2 FORM, where MODE2 deflexes to MODE1.

{6.3.1} deprocedured to MOID FORM:
MEEK procedure yielding MOID FORM.
(C3) The only meek (but not weak) coercion is dereferencing to STOWED (structured or rowed). This is clear from the rules:

\[ \{6.1.d\} \text{ weak REPETY STOWED FORM coercee:} \]
\[ \text{MEEK REPETY STOWED FORM,} \]
\[ \text{unless } \langle \text{MEEK} \rangle \text{ is } \langle \text{dereference to} \rangle \]
\[ \text{and } \langle \text{REPETY} \rangle \text{ is } \langle \text{EMPTY} \rangle . \]

5.4.4. Mode Ordering Theorem

**Mode Ordering Theorem**

Let a, b, and c be modes such that:

- a can be strongly coerced to c.
- a cannot be SORTLY coerced to c.
- b can be SORTLY coerced to c.
- c is not void.

Then \( h(a) < h(b) \).

**Proof of Mode Ordering Theorem**

*case 1 SORT=firm*

no strong coercion can occur before uniting \( (C1) \)
=> (since a can be strongly, -firmly coerced to c)

c cannot be a united mode

no meek coercion can follow uniting                   (C2)

=> (since b can be firmly coerced to non-united c)

b can be meekly coerced to c

=> this case reduces to SORT=meek

case 2 SORT=meek

a. c is \text{ref}[\ ]m

strong or firm coercions increase h by at least 1    (H1)

=>

\[ h(a) \leq h(c) - 1 \]

meek coercions to \text{ref}[\ ]m increase h by less than 1 (H1)

=>

\[ h(c) - 1 < h(b) \]

therefore, \( h(a) < h(b) \)

b. c is not \text{ref}[\ ]m

strong or firm coercions increase h by at least 2    (H1)

=>

\[ h(a) \leq h(c) - 2 \]

meek coercions to non \text{ref}[\ ]m increase h by less than 2 (H2)

=>
\[ h(c) - 2 < h(b) \]

therefore, \( h(a) < h(b) \)

case 3 SORT=weak

a. a can be strongly or firmly, but not meekly coerced to c

This case is the same as above.

b. a is meekly, but not weakly coerced to c

the one meek, \(-\)weak coercion is dereferencing to STOWED (C3)

\[ \Rightarrow \]

a=\text{pref} \ldots \text{pref ref } c \text{ and } c \text{ is STOWED}

\[ \Rightarrow \]

\[ h(a) \leq h(\text{ref } c) = h(c) - 1 \]

b can be weakly dereferenced and deprocedured to c

\[ \Rightarrow \]

b=\text{pref} \ldots \text{pref proc } c \text{ or } b=c

\[ \Rightarrow \]

\[ h(c) - 1 < h(b) \]

therefore, \( h(a) < h(b) \)

case 4 SORT=soft

h is unaffected by soft coercion (H3)

\[ \Rightarrow \]

\[ h(b) = h(c) \]

soft coercion is only to a reference to mode

\[ \Rightarrow \]
5.4.5. Proof of General Balancing Theorem

**General Balancing Theorem**

Let $m_1, ..., m_n$ be $n$ modes ($n \geq 1$).
Let $m$ in $\{m_1, ..., m_n\}$ be such that:
$h(m) > h(m_i)$ for all $i=1, ..., n$.
Then in any SORT balance of objects of modes $m_1, ..., m_n$ together with some hips and some displays, $m$ may be in the SORT position of the balance.

**Proof of General Balancing Theorem**

**Case** $\text{SORT}=\text{strong}$

$m$ may be in the strong position, since every position is a strong position.

**Case** $\text{SORT} \neq \text{strong}$

void can only be the mode of a balance in a strong position, therefore the coercion voiding is excluded for this case.
Assume \( m \) may not be in the \textsc{sort} position of the balance.

\[ \Rightarrow \text{(from the definition of balancing)} \]

There must be a mode \( p \) in \( \{m_1, \ldots, m_n\} \) that is in the \textsc{sort} position of the balance, and there must be a mode \( q \) such that:

- \( p \) can be \textsc{sortly} coerced to \( q \), and
- \( m \) can be strongly, but not \textsc{sortly} coerced to \( q \)

\[ \Rightarrow \text{(from the Mode Ordering Theorem)} \]

\[ h(m) < h(p) \]

but this is impossible because of the way in which \( m \) is chosen, therefore the assumption must be incorrect and \( m \) may be in the \textsc{sort} position of the balance.

5.5. Meek, Weak, and Soft Balancing Algorithms

As a result of the General Balancing Theorem, the following algorithm may always be used to find the mode in the \textsc{sort} position of a balance:

\begin{verbatim}
proc sort mode=(int num modes, [:] md m)md:
    begin
        int max:=0, position:=1;
        for i to num modes do
            if
                m[i]-a hip and
                m[i]-collateral modes and
                h of m[i]>max
                then
                    max:=h of m[i];
                    position:=i
            fi;
        m[position]
    end

Meek, weak, and soft balances are all slightly different.
\end{verbatim}
Depending on the SORT, the mode of the balance may be completely known, or only its type may be known. If only the type is known, then the mode of the balance must be returned by the balancing algorithm. The following table gives all types of balances:

<table>
<thead>
<tr>
<th>SORT of balance</th>
<th>type information</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>soft</td>
<td>reference to</td>
<td>lhs assignment, identity relation</td>
</tr>
<tr>
<td>weak</td>
<td>struct with FIELDS ROWS of MODE</td>
<td>selection, slice</td>
</tr>
<tr>
<td>meek</td>
<td>integral boolean UNITED proc with PARAMETERS</td>
<td>int CHOICE, bound boolean CHOICE UNITED CHOICE call</td>
</tr>
</tbody>
</table>

Table 5.2. Balancing

The following algorithm can be used for meek, weak, and soft balances. The type which is passed to this algorithm is the type demanded by the context of the balance, as outlined by the previous table.

```plaintext
proc m w s balance=
(int sort, type, num modes, [:] md m, ref md p)bool:
begin
# find sort position #
md sm:=sortmode(num modes,m);
bool can balance;
# get mode of balance in p #
can balance:=
case sort in
  soft coerce(sm,p),
  weak coerce(sm,p,type),
```
5.6. Operator Identification and Firm Balancing

When an applied occurrence of an operator indication is used in a formula, it must be determined which of the operator definitions is identified. To do this, ALGOL 68 requires that the operands of the operator must be firmly balanced to the corresponding mode in the definition. The General Balancing Theorem states a method of finding the firm position of the balance without knowing the mode of the balance. This method, together with the fact that the only firm positions in ALGOL 68 are the operands of formulas, can be used to speed up the identification of operators by combining the algorithms of firm balancing and operator identification. This avoids a repetition of finding the firm position of the balance for each operator definition that is tested.

The syntax of the Revised Report has changed the identification of operators slightly from what it was in the Report. A restriction, commonly referred to as the Dresden 2 condition can be stated, [9] page 4, as follows:

"If an applied occurrence A of an operator symbol S
identifies a defining occurrence $D$ in range $R$, then there shall be no other defining occurrence $D'$ for $S$ in any range $R'$ within $R$ and containing $A$ whose operand modes are related to those of $D$.

Here "related" is used in the same sense as the Report, i.e. two modes are related if both can be firmly coerced from the same mode. This would mean that in the following program, the applied occurrence of $\cdot\?\cdot$ would identify no defining occurrence, although according to the 1968 Report the first declaration of $\cdot\?\cdot$ would be identified.

```
begin
  op ? = (int a)int: skip;
begin
  op ? = (ref int a)int: skip;
  ? 1
end
end
```

The following algorithm identifies monadic operators (dyadic operators are handled similarly):

```
proc identify monadic operators=
  (ref node f, ref md a, [:] md m, int num modes) bool:
begin
  bool can identify:=false;
  deflist d:=monadic operator definitions[op of f];
  int t:=top of range stack;
  int definition range:=range of d;
  int active range:=range stack[t];
  md firm mode:=sortmode(num modes,m);

  # find a defining occurrence #
  while ¬can identify
  rep
    while active range>definition range
    rep
      t-:=1;
      if t=0 then goto errorexit fi;
      active range:=range stack[t]
    rep;
    while active range<definition range
    rep
```
d:=next of d;
if d:=:nil then goto errorexit fi;
definition range:=range of d
per:
while active range=definition range and ~can identify do
can identify:=coerce(firm,firm mode,md of d);
for i to num modes while can identify do
can identify:=coerce(strong,m[i],md of d);
per:
# check Dresden 2 #
deflist e:=monadic operator definitions[op of f];
bool related:=false;
while e:=:d and ~related 
rep
related:=check(md of e,md of d); e:=next of e
per:
if related then
  error("violation of Dresden 2 condition")
fi;

errorexit:
if can identify then
  op of f:=d; a:=delivered md of d
else
  error("cannot identify monadic operator")
fi;
can identify
end
6.1. Introduction

Coercion in a programming language may be defined as the transformation of one data type (or mode) into another. ALGOL 68 provides a large number of primitive data types, and allows a programmer to define any number of data types that may be built from these primitives. In addition, the orthogonal design of the language allows the programmer to use the data types he has defined in a large number of different contexts. This results in a complicated coercion process, and a number of different SORTs of coercion for use in different contexts. The SORTS: strong, firm, meek, weak and soft are ordered in the following way:
Figure 6.1

Meaning that all soft coercions are also weak coercions, and all weak coercions are meek coercions, etc. The following syntax of the Revised Report [6.1] shows this ordering of the SORTS of coercion:

**STRONG**: FIRM; widened to; rowed to; voided to.
**FIRM**: MEEEK; united to.
**MEEEK**: unchanged from; dereferenced to; deprocedured to.
**SOFT**: unchanged from; softly deprocedured to.

In these rules, FORM can be thought of as a construction (known as a coercend) to which the coercions apply. For ease of description, a pseudo coercion "hipping" has been invented to explain the transformations on MODES that take place in the following rules:

{5.5.2} strong MOID NEST skip: skip token.
{3.3.1.d} strong ROWS of MODE NEST collateral clause: EMPTY PACK.
{5.2.4} strong reference to MOST nihil: nil token.
{5.4.4} strong MOID NEST jump: goto token option,
It is also useful to distinguish between the two forms of rowing that may occur. The coercion "ref-rowing" is that which takes place when the "REFETY" in the following rule is "ref to":

{6.6} rowed to REFETY ROWS1 of MODE FORM:
where <ROWS1> is <row>,
STRONG REFLEXETY MODE FORM,
where <REFETY> is derived from <REFLEXETY>;
where <ROWS1> is <row ROWS2>,
STRONG REFLEXETY ROWS2 of MODE FORM,
where <REFETY> is derived from <REFLEXETY>;

The following table is a summary of what we consider to be coercions:

<table>
<thead>
<tr>
<th>strong</th>
<th>firm</th>
<th>meek</th>
<th>weak</th>
<th>soft</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>unite</td>
<td>dereference</td>
<td>dereference</td>
<td>dereference</td>
</tr>
<tr>
<td>widen</td>
<td>+ all meek</td>
<td>dereference</td>
<td>(restrict)</td>
<td></td>
</tr>
<tr>
<td>hip</td>
<td>coercions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>row</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ref-row</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ all firm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>coercions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1. Coercion

The next table illustrates where in the program the coercions of a particular sort are initiated:
6.2. The Zosel Strong-Firm Coercion Algorithm

The Zosel algorithm [12] for strong and firm coercion is described by Peck [5]. This description, and the included model (together with the improved model of Kwan [4]) were used as a basis for the Vancouver implementation. Peck describes the problem of coercion as a problem of finding a path ("coercion sequence") through a graph from one node ("the a priori mode") to another ("the a posteriori mode"). This graph has as its nodes the modes of the program, and as its arcs the "coercion steps" of the language.

The coercion algorithm searches down two trees in this graph ("the a posteriori route" and "the a priori route"), which have as their roots the "aposteriori" and "a priori" modes. If
these two trees meet somewhere, then a coercion path has been found. If they do not, then no coercion path exists between the two modes. Marking the a posteriori route ("posting") is done first. Then the apriori route is followed until a marked node is encountered. The coercion path can then be easily picked up, since a marked node always contains a link back to its parent in the tree.

```plaintext
proc coerce=(int sort, ref md apriori, apost,ref cs c)bool:
begin
  md m; bool can coerce:=false;
  post(sort,apost);
  m:=apriori;
  a:
    if flag[class of m] then
      can coerce:=true;
      finish
    fi;
    if modetype of m = proc then
      link of m:=submode of m;
      coercion of m:=deproc;
      m:=submode of m; goto a
    fi;
    if modetype of m=ref then
      link of m:=submode of m;
      coercion of m:=deref;
      m:=submode of m; goto a
    fi;
  fi;
  finish:
  if can coerce then
    c:=buildcoercionseq(apriori,apost)
  fi;
  can coerce
end

proc post=(int sort,ref md apost)void:
begin
  md a,b;
  case sort in
  # firm #
    if apost=void then
      flag[class of apost]:=true
    fi;
    if modetype of apost=united then
      for i to numberofsubmodes of apost rep
```
6.3. Meek, Weak, and Soft Coercion

Soft coercion involves deproceduring a given mode until a reference to mode is found:
proc soft coerce= (ref md apriori, apost, ref cs c) bool:
begin
    md m;
apost:=apriori;
    while modetype of apost=procedure do
    begin
        m:=submode of apost;
        link of apost:=m;
        coercion of apost:=deproc;
        apost:=m
    end;
    if modetype of apost=reference then
        c:=buildcoercionseq (apriori, apost); true
    else
        false
    fi
end

For weak coercion, the given mode is deproced and dereferenced as long as possible. The resulting mode should be of a given type (either row, rowof, or structured) or reference to this given type:

proc weak coerce=
(ref md apriori, apost, int t, ref cs c) bool:
begin
    md m;
a:
apost:=apriori;
    while modetype of apost=procedure do
    begin
        m:=submode of apost;
        link of apost:=m; coercion of apost:=deproc;
        apost:=m
    end;
    if modetype of apost=reference then
        while m:=submode of apost; modetype of m=reference do
        begin
            link of apost:=m;
            coercion of apost:=deref;
            apost:=m
        end;
    else
        m:=apost
    fi;
    if modetype of m=procedure then goto a fi;
if modetype of m=t then
  c:=buildcoercionseq(apriori,apost); true
else
  false
fi
end

There are two types of meek coercion: one in which the a posteriori mode is given, and one in which it is known that the a posteriori mode is either of type union or of type procedure with parameters. In either case, the given mode is dereferenced or deprocedured as long as possible:

proc coerce meek unknown=
(ref md apriori, apost, int type, ref cs c)bool:
begin
  md m; int t;
  apost:=apriori;
  while t:=modetype of apost; t=reference or t=procedure
    rep
      m:=submode of apost;
      link of apost:=m;
      coercion of apost:=(t=reference|deref|deproc);
      apost:=m
    end
  if modetype of apost=type then
    c:=buildcoercionseq(apriori,apost); true
  else
    false
  fi
end

proc coerce meek known=
(ref md apriori, apost, ref cs c)bool:
begin
  md m,a; int t;
  a:=apriori;
  while t:=modetype of a; t=reference or t=procedure
    rep
      m:=submode of a;
      link of a:=m;
      coercion of a:=(t=reference|deref|deproc);
      a:=m
    end
  if a=apost then
    c:=buildcoercionseq(apriori,apost); true
  else
    false
  fi
6.4. Examples of Coercion Paths

Some examples of the a posteriori and a priori trees through the mode table follow. The method of illustrating these trees and the first example are adapted from Peck [5].

The first example shows the lengthening coercions (rowing and uniting) in the a posteriori tree, and the shortening coercions (dereferencing and deproceduring) in the a priori tree. The trees meet at the mode int:

aposteriori mode: \texttt{[ ] union(int,real)}

a priori mode: \texttt{proc ref int}

a posteriori tree:

```
[ ] union(int,real)
  | (rowing)
  union(int,real)
  | (uniting)
  int
  real
```

a priori tree:

```
proc ref int
  | (deproceduring)
  ref int
  | (dereferencing)
```
The following example shows the necessity for the "reverse ravelling" of united modes. "Reverse ravelling" consists of placing pointers in a united mode to all other united modes in the program whose submodes are a subset of the given united mode. For example, a pointer to union(int,real) would be placed in the mode union(int,real,Bool). During the posting part of coercion, this pointer is needed in order to flag union(int,real) when the a posteriori mode is union(int,real,Bool):

\begin{figure}
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Example of "reverse ravelling".}
\end{figure}

This example illustrates the (somewhat strange) coercion "ref-rowing". Note also in this example that the a priori tree
is only followed until a flagged mode is found, even though the mode \texttt{ref real} can still be "dereferenced".

a posteriori mode: \texttt{ref [], real}
a priori mode: \texttt{ref ref real}
a posteriori tree:

\begin{verbatim}
\texttt{ref [], real}
\texttt{ref [ ] real}
\texttt{ref real}
\end{verbatim}

a priori tree:

\begin{verbatim}
\texttt{ref ref real}
\texttt{ref real}
\end{verbatim}

Figure 6.4

6.5. Lengthening and Shortening

The authors and editors of ALGOL 68 have put much effort into allowing a programmer to write various constructions as briefly as possible. For example, the conditional clause \texttt{if x>0 then x else 0 fi} could also be written as \texttt{(x>0|x=0)}, and the serial clause \texttt{begin int i; read(i); i=0 end} could be written as \texttt{((int i; read(i); i=0). Coercions are another mechanism that allows the programmer to write his program
briefly. \texttt{real } \texttt{x:=real (i)} \ may \ be \ written \ \texttt{real } \texttt{x:=i}. \ Why \ the \ programmer \ cannot \ also \ write:

\begin{itemize}
  \item \texttt{long real } \texttt{x:=1} \ instead \ of \ \texttt{long real } \texttt{x:=long 1}, \ or
  \item \texttt{[1:99]short int } \texttt{a=(1,...,99)} \ instead \ of \ \texttt{[1:99]short int } \texttt{a=(short 1,...,short 99)}, \ or
  \item \texttt{short int } \texttt{s; for } \texttt{i} \texttt{ to } \texttt{n do } \texttt{s+:=i}; \ instead \ of
  \item \texttt{short int } \texttt{s; for } \texttt{i} \texttt{ to } \texttt{n do } \texttt{s+:=shorten i}
\end{itemize}

is one of the greatest surprises in the language.

These constructions could be allowed if the two coercions lengthening and shortening were included in the language. The justification for excluding them is a possible, undesired loss of precision when an unknowing programmer writes such as \texttt{long real } \texttt{x:=2.1} when he should have written \texttt{long real } \texttt{x:=long 2.1}. Another is some possible ambiguities that may arise e.g., \texttt{if true then 1 else long 1 fi}. The Vancouver implementation allows these coercions, but warns the programmer that they are not in the language, and that they may result in a loss of precision.

The lengthening and shortening coercions are only allowed in a strong position and are restricted in the following ways:

1. No lengthening or shortening may follow widening.
2. No shortening may follow lengthening.
3. No lengthening may follow shortening.

These restrictions eliminate ambiguous coercion paths and
guarantee maximum precision in the coercion. For example, the path:

\[
\text{int} \rightarrow \text{long} \quad \text{int} \rightarrow \text{w} \rightarrow \text{long} \quad \text{real}
\]

is allowed, but the path

\[
\text{int} \rightarrow \text{w} \rightarrow \text{real} \rightarrow \text{l} \rightarrow \text{long} \quad \text{real}
\]

is not, since the lengthening coercion follows the widening coercion. The latter path is less desirable, since precision may be lost due to the inexact representation of a floating point number in a computer.

The lengthening and shortening coercions in the Vancouver compiler are implemented in a manner suggested by the following rules:

\[
\begin{align*}
\text{STRONG} & : \text{FIRM}; \text{widened to}; \text{rowed to}; \\
& \quad \text{voided to}; \text{shortened to}; \text{lengthened to}.
\end{align*}
\]

\[
\begin{align*}
\text{SLEEK} & : \text{MEEK}; \text{shortened to}; \\
& \quad \text{lengthened to}.
\end{align*}
\]

\[
\text{widened to SIZETY real FORM:} \\
\quad \text{SLEEK SIZETY integral FORM.}
\]

\[
\text{widened to SIZETY structured with letter r letter e has SIZETY real field letter i letter m has SIZETY real field FORM:} \\
\quad \text{SLEEK SIZETY real FORM;}
\]

\[
\text{widened to SIZETY real FORM.}
\]

\[
\text{lengthened to MODE FORM:} \\
\quad \text{where <MODE> is <long MODE1>,}
\]

\[
\text{lengthened to MODE1 FORM or}
\]

\[
\text{alternatively MEEK MODE1 FORM;}
\]

\[
\text{unless MODE begins with long,}
\]

\[
\text{lengthened to short MODE FORM or}
\]

\[
\text{alternatively MEEK short MODE FORM.}
\]

\[
\text{shortened to MODE FORM:} \\
\quad \text{where <MODE> is <short MODE1>,}
\]

\[
\text{shortened to MODE1 FORM or}
\]

\[
\text{alternatively MEEK short MODE FORM.}
\]
shortened to MODE1 FORM or
alternatively MEEK MODE1 FORM;
unless MODE begins with short,
shortened to long MODE FORM or
alternatively MEEK long MODE FORM.
Conclusion

This thesis has discussed some problems of compiling ALGOL 68 programs that are concerned with coercion, balancing and operator identification. Solutions that are being used successfully in the Vancouver compiler were presented.

The purpose and nature of a data structure called the Program Tree was explained. A brief discussion of how the Program Tree is built was followed by a detailed explanation of how the identification and mode checking routines of an ALGOL 68 compiler interact with it.

The history of balancing was traced, and a general method of balancing (any SORT) was given and shown to be correct.

The coercion models of Peck and Kwan were adapted so that they can be used in an ALGOL 68 compiler. Some extended coercions were implemented that may ease some of the writing burden off the programmer.
Bibliography


Appendix 1. Sample Runs

A number of program examples showing the program tree after the mode checking pass follow. The coercion sequences that are associated with a program tree node can be seen. Also it can be seen that operators and identifiers have been identified. The output has been taken directly from the compiler and squashed so that it will fit on one page.
```
begin
  mode a = proc ref int;
  a x = skip; a y, z;
  case 0 in
    x, y, z, # DEREFERENCED, DEPROCEDURED #
    out # DEPROCEDURED #
  esac := ~x := 1 # ASSIGNATION IS DEREFERENCED #
end
```

0.48 SECONDS FOR PASS 1
0.01 SECONDS FOR PASS 2
PARSE SUCCESSFUL
0.34 SECONDS FOR PASS 3

PROGRAM TREE:
| SERIAL CLAUSE
| DECLARATION PROLOGUE
| MODE DECLARATION
| A
| MODE 98
| TAG DECLARATION LIST
| TAG DECLARATION
| MODE 98
| X
| SKIP
| COERCION SEQUENCE: HIP, TO MODE 98
| TAG DECLARATION LIST
| TAG DECLARATION
| MODE 99
| Y
| --
| TAG DECLARATION
| MODE 99
| Z
| --
| ASSIGNATION
| COERCION SEQUENCE: VOID, TO MODE 18
| CASE CLAUSE
| DENOTATION
| COERCION SEQUENCE:
|TAG |
|COERCION SEQUENCE: DEREF, DEPROC, TO MODE 23 |
|TAG DECLARATION |
|MODE 99 |
|Z |

|TAG |
|COERCION SEQUENCE: DEPROC, TO MODE 23 |
|TAG DECLARATION |
|MODE 98 |
|X |

|SKIP |
|COERCION SEQUENCE: HIP, TO MODE 20 |

|TAG |
|COERCION SEQUENCE: DEREF, DEPROC, TO MODE 23 |
|TAG DECLARATION |
|MODE 99 |
|Y |

|SKIP |
|COERCION SEQUENCE: HIP, TO MODE 20 |

|ASSIGNATION |
|COERCION SEQUENCE: DEREF, TO MODE 1 |

|TAG |
|COERCION SEQUENCE: DEPROC, TO MODE 23 |
|TAG DECLARATION |
|MODE 98 |
|X |

|SKIP |
|COERCION SEQUENCE: HIP, TO MODE 98 |

|DENOTATION |
|COERCION SEQUENCE: |
|1 |

0.48 SECONDS FOR PASS 4
0.00 SECONDS FOR PASS 5
1.31 SECONDS FOR TOTAL COMPILATION(S)
begin

# WEAK COERCION AND BALANCING #

# SLICES #

(/1:2/) real x; (/1:2/) real y = skip;

case 0 in

X, # DEREFERENCE #
y, # NO COERCION #
(1,2) # WIDEN 1 AND 2 #

out

skip # HIP #
end

# STRUCTURES #

struct (real i,j)a = skip;

i of case 0 in

a, # NO COERCION #
(1,2) # WIDEN 1 AND 2 #

out

skip # HIP #

end

0.42 SECONDS FOR PASS 1
0.02 SECONDS FOR PASS 2
PARSE SUCCESSFUL
0.33 SECONDS FOR PASS 3

PROGRAM TREE:
|SERIAL CLAUSE

|DECLARATION PROLOGUE

|TAG DECLARATION LIST

|TAG DECLARATION

|MODE 99

X

|----|

|TAG DECLARATION LIST

|TAG DECLARATION

|MODE 100

Y

|SKIP

|COERCION SEQUENCE: HIP, TO MODE 100

|SERIES WITH DEFS

|SLICE

|COERCION SEQUENCE: VOID, TO MODE 18

|CASE CLAUSE

|DENOTATION
UNIT LIST
| TAG
| COERCION SEQUENCE: DEREF, TO MODE 100
| TAG DECLARATION
| MODE 99
| X
| ----
| TAG
| COERCION SEQUENCE:
| TAG DECLARATION
| MODE 100
| Y
| SKIP
| COERCION SEQUENCE: HIP, TO MODE 100
| COLLATERAL CLAUSE
| DENOTATION
| COERCION SEQUENCE: WIDEN, TO MODE 3
| 1
| DENOTATION
| COERCION SEQUENCE: WIDEN, TO MODE 3
| 2
| SKIP
| COERCION SEQUENCE: HIP, TO MODE 100
| TRIMSCRIPT LIST
| TRIMSCRIPT
| DENOTATION
| COERCION SEQUENCE:
| 1
| DENOTATION
| COERCION SEQUENCE:
| 2
| ----
| TAG DECLARATION LIST
| TAG DECLARATION
| MODE 102
| a
| SKIP
| COERCION SEQUENCE: HIP, TO MODE 102
| SELECTION
| COERCION SEQUENCE: VOID, TO MODE 18
| CASE CLAUSE
| DENOTATION
| COERCION SEQUENCE:
| 0
| UNIT LIST
| TAG
| COERCION SEQUENCE:
| TAG DECLARATION
| MODE 102
|A
|SKIP
|COERCION SEQUENCE: HIP, TO MODE 102
|COLLATERAL CLAUSE
|DENOTATION
|COERCION SEQUENCE: WIDEN, TO MODE 3
| 1
|DENOTATION
|COERCION SEQUENCE: WIDEN, TO MODE 3
| 2
|SKIP
|COERCION SEQUENCE: HIP, TO MODE 102

0.49 SECONDS FOR PASS 4
0.00 SECONDS FOR PASS 5
1.26 SECONDS FOR TOTAL COMPILATION(S)
begin
  int x, real y;
  case 0 in
    x, # DEREFERENCE, WIDEN #
    y, # DEREFERENCE #
    1 # WIDEN #
  esac + # IDENTIFY OP (REAL, REAL) REAL #
  if true then
    1.0 # NO COERCION #
  else
    skip # HIP #
  fi
end

0.48 SECONDS FOR PASS 1
0.01 SECONDS FOR PASS 2
PARSE SUCCESSFUL
0.29 SECONDS FOR PASS 3

PROGRAM TREE:
| SERIAL CLAUSE
  | DECLARATION LIST
    | TAG DECLARATION LIST
      | TAG DECLARATION
        | MODE 23
        | x
        | ----
    | TAG DECLARATION LIST
      | TAG DECLARATION
        | MODE 25
        | y
        | ----
| DYADIC FORMULA
| ICOERCION SEQUENCE: VOID, TO MODE 18
| OP DECLARATION
  | MODE 3
  | MODE 3
  | MODE 3
  | +
  | ----
| CASE CLAUSE
  | DENOTATION
    | ICOERCION SEQUENCE:
      | 0
| UNIT LIST
TAG
| COERCION SEQUENCE: DEREF, WIDEN, TO MODE 3
| TAG DECLARATION
| MODE 23
| X
|----

TAG
| COERCION SEQUENCE: DEREF, TO MODE 3
| TAG DECLARATION
| MODE 25
| Y
|----

DENOTATION
| COERCION SEQUENCE: WIDEN, TO MODE 3
| 1

SKIP
| COERCION SEQUENCE: HIP, TO MODE 20
| CONDITIONAL CLAUSE
| DENOTATION
| COERCION SEQUENCE:
| TRUE
| DENOTATION
| COERCION SEQUENCE:
| +0.100000000000000e+01
| SKIP
| COERCION SEQUENCE: HIP, TO MODE 3

0.49 SECONDS FOR PASS 4
0.00 SECONDS FOR PASS 5
1.27 SECONDS FOR TOTAL COMPILATION(S)
begin

# IDENTIFICATION ERRORS #

# UNDEFINED IDENTIFIER #

aleph;

# MULTIPLY DEFINED IDENTIFIER #

int i; real i;

# UNDEFINED OPERATOR #

+ bool(skip);

# VIOLATION OF DRESDEN 2 #

op += (ref int a)int: skip; +1;

# UNDEFINED FIELD SELECTOR #

struct(int i, j) x; k of x;

skip

end

0.44 SECONDS FOR PASS 1
0.02 SECONDS FOR PASS 2

<ERROR: 302><SEV:9> 2 1 THE IDENTIFIER 'I' IS MULTIPLY DEFINED.
THE FIRST DEFINITION IS USED.

PARSE SUCCESSFUL
0.43 SECONDS FOR PASS 3

<ERROR: 401><SEV:8><COOR:0><CRRL:1> THE IDENTIFIER 'ALEPH' IS UNDEFINED.

<ERROR: 402><SEV:8><COOR:3><CRRL:1> THE OPERATOR '++' IS UNDEFINED.

<ERROR: 430><SEV:8><COOR:5><CRRL:1> THE OPERATOR '++' CANNOT BE IDENTIFIED, SINCE THE DRESDEN CONDITION IS VIOLATED.

<ERROR: 403><SEV:8><COOR:8><CRRL:1> 'K OF X' IS AN INVALID SINCE SELECTION, SINCE THERE IS NO FIELD SELECTOR 'K' IN 'REF STRUCT(INT I,INT J)'.

0.07 SECONDS FOR PASS 4
0.96 SECONDS FOR TOTAL COMPILATION(S)
begin
  # BALANCING ERRORS #
  # SOFT #
  int i: (1|i:=1,1):=2;
  # WEAK TO ROW #
  (/1:6/)int a; (true|a|1.0) (/1/);
  # WEAK TO STRUCTURE #
  struct (int i,j)x; i of (true|x|1);
  # MEEK TO BOOL #
  if (true|1|skip) then skip fi;
  # MEEK TO UNITED #
  case (true|1|skip) in (int h):1,(real i):2
  esac;
end

0.41 SECONDS FOR PASS 1
0.02 SECONDS FOR PASS 2
PARSE SUCCESSFUL
0.36 SECONDS FOR PASS 3

<ERROR: 406><SEV:8><COOR:2><CRRL:1> THESE MODES OF AN ASSIGNMENT CANNOT BE BALANCED TO A 'REF':
  'INT' (FROM '1')
  'REF INT' (FROM 'I:=1')
  'SKIP' (FROM 'SKIP')

<ERROR: 407><SEV:8><COOR:4><CRRL:1> INVALID SLICE. THESE MODES CANNOT BE WEAKLY BALANCED TO A ROW:
  'REAL' (FROM '+1.0')
  'REF ()INT' (FROM 'A')

<ERROR: 408><SEV:8><COOR:7><CRRL:1> THESE MODES CANNOT BE WEAKLY BALANCED TO A STRUCTURE:
  'INT' (FROM '1')
  'REF STRUCT(INT I,INT J)' (FROM 'X')

<ERROR: 410><SEV:8><COOR:8><CRRL:1> THESE MODES CANNOT BE MEEKLY BALANCED TO 'BOOL':
  'SKIP' (FROM 'SKIP')
  'INT' (FROM '1')

<ERROR: 411><SEV:8><COOR:9><CRRL:1> THESE MODES CANNOT BE MEEKLY BALANCED TO A UNION:
  'SKIP' (FROM 'SKIP')
'INT' (FROM '1')

0.19 SECONDS FOR PASS 4
0.00 SECONDS FOR PASS 5
0.98 SECONDS FOR TOTAL COMPILATION(S)
COORD  CRRL

COUNT

0     0 |'PR LIST,AMS,dartmouth PR

 *  * |begin
 *  * | # COERCION ERRORS #

 *  * | # INVALID CALL OR SLICE #
 *  * | struct(int i,j) x; x(1);

 3     0 | # INCORRECT DISPLAY #

 6     0 | int e; e:=(1,2);

 8     0 | # INVALID STRONG COERCION #

 12    0 | int b; b:=1.0;

 12    0 | # INVALID WEAK COERCION #

 12    0 | (1:10)int a; i of a; (x:=(10,1)) (/1/);

 0.45 SECONDS FOR PASS 1
 0.02 SECONDS FOR PASS 2

PARSE SUCCESSFUL

0.30 SECONDS FOR PASS 3

<ERROR: 404><SEV:8><COORD:2><CRRL:1> 'X' CANNOT BE THE PRIMARY OF A CALL OR SLICE, SINCE IT IS OF MODE 'REF STRUCT(INT I,INT J)'.

<ERROR: 426><SEV:8><COORD:4><CRRL:1> THE DISPLAY '(1,2)' CANNOT BE OF MODE 'INT'.

<ERROR: 416><SEV:8><COORD:7><CRRL:1> '+1.0' (OF MODE 'REAL') CANNOT BE STRONGLY COERCED TO 'INT'.

<ERROR: 418><SEV:8><COORD:9><CRRL:1> 'A' CANNOT BE THE SECONDARY OF A SELECTION, SINCE ITS MODE, 'REF ()INT', CANNOT BE WEAKLY COERCED TO A STRUCTURE.

<ERROR: 419><SEV:8><COORD:11><CRRL:1> 'X:=(10,1)' CANNOT BE THE PRIMARY OF A SLICE, SINCE ITS MODE, 'REF STRUCT(INT I,INT J)', CANNOT BE WEAKLY COERCED TO A ROW.

0.17 SECONDS FOR PASS 4
0.00 SECONDS FOR PASS 5
0.94 SECONDS FOR TOTAL-compilation(S)
COORD   CRRL
COUNT.

0   0 | **PR LIST,AMS,dartmouth pr**
   -   - | **begin**
   -   - | **# BAD COUNTS #**
   -   - | **PROC p = (real u,v) void:skip: p(1,2,3);**
   -   - | **# INCORRECT NUMBER OF ARGUMENTS #**
   -   - | **PROC p = (real u,v) void:skip: p(1,2,3);**
   -   - | **# INCORRECT NUMBER OF TRIMSCRIPTS #**
   -   - | **PROC p = (real u,v) void:skip: p(1,2,3);**
   -   - | **# INCORRECT DISPLAY LENGTH #**
   -   - | **STRUCT(int i,j) x; x:=(1,2,3);**
   -   - | **end**
   -   - | **skip**

0.38 SECONDS FOR PASS 1
0.01 SECONDS FOR PASS 2
PARSE SUCCESSFUL
0.29 SECONDS FOR PASS 3

<ERROR: 421><SEV:8><COOR:2><CRRL:1> THE CALL 'P(1,2,3)' HAS AN INCORRECT NUMBER OF ACTUAL PARAMETERS.

<ERROR: 423><SEV:8><COOR:7><CRRL:1> THERE ARE TOO FEW TRIMSCRIPTS IN THE SLICE 'D(1:2)'.

<ERROR: 422><SEV:8><COOR:8><CRRL:1> THERE ARE TOO MANY TRIMSCRIPTS IN THE SLICE 'D(1:2,3:4,5:6)'.

<ERROR: 427><SEV:8><COOR:13><CRRL:1> THE STRUCTURE DISPLAY '('1,2,3)' HAS AN INCORRECT NUMBER OF FIELDS.

0.19 SECONDS FOR PASS 4
0.00 SECONDS FOR PASS 5
0.87 SECONDS FOR TOTAL COMPILATION(S)
begin  
  # ERROR RECOVERY #  
  int i=skip; (1:3) real y; int z=2;  
  i:=(skip+y/1,2/) | z:=2;  
  # LIMIT ERROR MESSAGES #  
  (1:10) int a:=(1.0,2.0,3.0,  
               4.0,5.0,6.0,  
               7.0,8.0,9.0,10.0);  
  skip  
end

0.36 SECONDS FOR PASS 1  
0.02 SECONDS FOR PASS 2

PARSE SUCCESSFUL  
0.35 SECONDS FOR PASS 3

<ERROR: 417><SEV:8><COOR:3><CRRL:1> 'I' CANNOT BE ON THE  
LEFT HAND SIDE OF AN ASSIGNATION, SINCE  
ITS MODE, 'INT', DOES NOT BEGIN WITH 'REF'.

<ERROR: 415><SEV:8><COOR:3><CRRL:1> 'SKIP' (OF MODE 'SKIP')  
CANNOT BE MEANLY  
COERCED TO 'BOOL'.

<ERROR: 422><SEV:8><COOR:3><CRRL:1> THERE ARE TOO MANY  
TRIMSCRIPTS IN THE SLICE 'Y(1,2)'.

<ERROR: 417><SEV:8><COOR:4><CRRL:1> 'Z' CANNOT BE ON THE  
LEFT HAND SIDE OF AN ASSIGNATION, SINCE  
ITS MODE, 'INT', DOES NOT BEGIN WITH 'REF'.

<ERROR: 416><SEV:8><COOR:4><CRRL:1> '+'1.0' (OF MODE 'REAL')  
CANNOT BE STRONGLY COERCED TO 'INT'.

<ERROR: 416><SEV:8><COOR:5><CRRL:1> '+'2.0' (OF MODE 'REAL')  
CANNOT BE STRONGLY COERCED TO 'INT'.

<ERROR: 416><SEV:8><COOR:6><CRRL:1> '+'3.0' (OF MODE 'REAL')  
CANNOT BE STRONGLY COERCED TO 'INT'.

<NOTE><SEV:0><COOR:15><CRRL:1> THE PREVIOUS ERROR  
HAS OCCURRED 7 MORE TIMES.

0.32 SECONDS FOR PASS 4  
0.00 SECONDS FOR PASS 5  
1.05 SECONDS FOR TOTAL COMPILATION(S)
Appendix 2. Program Tree Nodes

The following are all of the program tree nodes that may be built. The grammar from which they are derived is given in [10].
### Declarer.1  Temporary

<table>
<thead>
<tr>
<th>temp</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>mn</td>
</tr>
<tr>
<td>cw</td>
<td>h</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
</tbody>
</table>

> mode declaration

### Declarer.2  Primitive

<table>
<thead>
<tr>
<th>prim</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>mn</td>
</tr>
<tr>
<td>cw</td>
<td>h</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
<tr>
<td>acc #</td>
<td></td>
</tr>
</tbody>
</table>
### Declarer.3 Reference

<table>
<thead>
<tr>
<th>ref</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>mn</td>
</tr>
<tr>
<td>cw</td>
<td>h</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
</tbody>
</table>

→ submode

### Declarer.4 Rowof

<table>
<thead>
<tr>
<th>rowof</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>mn</td>
</tr>
<tr>
<td>cw</td>
<td>h</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
</tbody>
</table>

→ pair list or nil

→ > submode

→ > ref submode

strict flex either
Declarer.5  Row

```
<table>
<thead>
<tr>
<th>row</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>mn</td>
</tr>
<tr>
<td>cw</td>
<td>h</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-&gt; pair list or nil</td>
</tr>
<tr>
<td></td>
<td>-&gt; submode</td>
</tr>
<tr>
<td></td>
<td>-&gt; ref submode</td>
</tr>
<tr>
<td>strict flex either</td>
<td></td>
</tr>
</tbody>
</table>
```

Declarer.6  Union

```
<table>
<thead>
<tr>
<th>union</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>mn</td>
</tr>
<tr>
<td>cw</td>
<td>h</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-&gt; submode 1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-&gt; submode n</td>
</tr>
</tbody>
</table>
```
Declarer.7  Struct

<table>
<thead>
<tr>
<th>struct</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>mn</td>
</tr>
<tr>
<td>cw</td>
<td>h</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;-field 1</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>&gt;-field n</td>
</tr>
</tbody>
</table>

Declarer.8  Proc Type

<table>
<thead>
<tr>
<th>proc</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>mn</td>
</tr>
<tr>
<td>cw</td>
<td>h</td>
</tr>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;-submode</td>
</tr>
</tbody>
</table>
Declarer.9      Procp

\[
\begin{array}{|c|c|}
\hline
\text{procp} & \text{n+1} \\
\hline
\text{dn} & \text{mn} \\
\hline
\text{cw} & \text{h} \\
\hline
\text{flags} \\
\hline
\text{link} \\
\hline
\end{array}
\]

Declarer.10      Field

\[
\begin{array}{|c|c|}
\hline
\text{field} & \text{4} \\
\hline
\text{dn} & \text{mn} \\
\hline
\text{cw} & \text{h} \\
\hline
\text{flags} \\
\hline
\text{link} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{tag acc #} \\
\hline
\end{array}
\]
Coercend. 1  Routine

<table>
<thead>
<tr>
<th>routine</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
</tbody>
</table>

- coercion sequence
- formal p. p. or nil
- unit mode
- "unit"
- routine mode

Coercend. 2  Assignation

<table>
<thead>
<tr>
<th>assig</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
</tbody>
</table>

- coercion sequence
- "tertiary"
- "unit"
Coercend. 3  Identity Is Relation

ident is | 3
range # | coor #

• ——> coercion sequence
• ——> tertiary 1
• ——> tertiary 2

Coercend. 4  Identity Is Not Relation

ident isnt | 3
range # | coor #

• ——> coercion sequence
• ——> tertiary 1
• ——> tertiary 2

Coercend. 5  Cast

cast | 3
range # | coor #

• ——> coercion sequence
• ——> mode of cast
• ——> unit
Coercend.6  Dyadic Formula

<table>
<thead>
<tr>
<th>formula</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td>→ coercion sequence</td>
</tr>
<tr>
<td>op acc #</td>
<td>(--&gt; op dec)</td>
</tr>
<tr>
<td>•</td>
<td>→ left operand</td>
</tr>
<tr>
<td>•</td>
<td>→ right operand</td>
</tr>
</tbody>
</table>

Coercend.7  Monadic Formula

<table>
<thead>
<tr>
<th>monadic</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td>→ coercion sequence</td>
</tr>
<tr>
<td>op acc #</td>
<td>(--&gt; op dec)</td>
</tr>
<tr>
<td>•</td>
<td>→ &quot;operand&quot;</td>
</tr>
</tbody>
</table>

Coercend.8  Local Generator

<table>
<thead>
<tr>
<th>loc gen</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td>→ coercion sequence</td>
</tr>
<tr>
<td>•</td>
<td>→ mode of generator</td>
</tr>
</tbody>
</table>
Coercend.9  Global Generator

| glo gen | 2 |
| range # | coor # |

• --- > coercion sequence

• --- > mode of generator

Coercend.10  Selection

| select | 4 |
| range # | coor # |

• --- > coercion sequence

tag acc # (-->field mode)

• --- > "secondary"

selection mode

Coercend.11  Goto

| goto | 2 |
| range # | coor # |

• --- > coercion sequence

tag acc # (-->tag declaration)
Coercend. 12  Tag

```
tag  |  2
range # | coor #

--- coerced sequence
```

Coercend. 13  Skip

```
skip  |  1
range # | coor #

--- coerced sequence
```

Coercend. 14  Nil

```
nil  |  1
range # | coor #

--- coerced sequence
```
### Coercend.15 Vacuum

<table>
<thead>
<tr>
<th>vacuum</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Coercend.16 Known Denotation

<table>
<thead>
<tr>
<th>k deno</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>deno acc #</td>
<td></td>
</tr>
</tbody>
</table>

### Coercend.17 Slice

<table>
<thead>
<tr>
<th>slice</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>slice mode</td>
<td></td>
</tr>
</tbody>
</table>

Coercend.18  Calice

<table>
<thead>
<tr>
<th>calice</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>——&gt; coercion sequence</td>
</tr>
<tr>
<td></td>
<td>——&gt; &quot;primary&quot;</td>
</tr>
<tr>
<td></td>
<td>——&gt; trimment list</td>
</tr>
<tr>
<td>calice mode</td>
<td></td>
</tr>
</tbody>
</table>

Coercend.19  Call

<table>
<thead>
<tr>
<th>call</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>——&gt; coercion sequence</td>
</tr>
<tr>
<td></td>
<td>——&gt; &quot;primary&quot;</td>
</tr>
<tr>
<td></td>
<td>——&gt; argument list</td>
</tr>
<tr>
<td>call mode</td>
<td></td>
</tr>
</tbody>
</table>
### Structural.20 Label Sequence

<table>
<thead>
<tr>
<th>label seq</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td>tag acc #1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>tag acc #n</td>
<td></td>
</tr>
</tbody>
</table>

### Structural.21 Program

<table>
<thead>
<tr>
<th>program</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
</tbody>
</table>

---

| •---label seq. or nil |
| •---"enclosed clause" |
### Structural.22  Loop

<table>
<thead>
<tr>
<th>loop</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
</tbody>
</table>

- `tag declaration` (from)
- `unit" (by)`
- "unit" (to)
- serial clause
- "unit" (do)

---

### Structural.23  Trimmer

<table>
<thead>
<tr>
<th>trimmer</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
</tbody>
</table>

- "bound" 1 or nil
- "bound" 2 or nil
- "bound" 3 or nil
Structural.24 Pair

<table>
<thead>
<tr>
<th>pair</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>→ &quot;bound&quot; 1 or nil</td>
</tr>
<tr>
<td></td>
<td>→ &quot;bound&quot; 2 or nil</td>
</tr>
</tbody>
</table>

Structural.25 Parameters List

<table>
<thead>
<tr>
<th>par list</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>→ tag dec 1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>→ tag dec n</td>
</tr>
</tbody>
</table>

Structural.26 Collateral Clause

<table>
<thead>
<tr>
<th>coll cl</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>→ &quot;unit&quot; 1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>→ &quot;unit&quot; n</td>
</tr>
</tbody>
</table>
Structural.27  Conditional Clause

<table>
<thead>
<tr>
<th>cond cl</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-> "serial clause" 1
-> "serial clause" 2
-> "serial clause" 3

Structural.28  Case Clause

<table>
<thead>
<tr>
<th>case cl</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-> "serial clause a"
-> "unit list"
-> "serial clause b"

Structural.29  Conformity Case Clause

<table>
<thead>
<tr>
<th>conf cl</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-> "serial clause a"
-> "alternative list"
-> "serial clause b"
Structural.30 Alternative

```
+----------+---+
| alter    | 3 |
| range #  | coor # |
|          |     |
|          |     | --> mode
|          |     | --> tag or nil
|          |     | --> "unit"
```

Structural.31 Serial Clause

```
+----------+---+
| serial cl| 2 |
| range #  | coor # |
|          |     |
|          |     | --> declaration prologue
|          |     | --> parade
```

Structural.32 Train

```
+----------+---+
| train    | n |
| range #  | coor # |
|          |     |
|          |     | --> unit series 1
|          |     | ... |
|          |     | --> unit series n
```
Structural.33  Unit Series

```
unit ser | n
range # | coor #

| .-------|"unit" 1 |
| . . .   |
| .-------|"unit" n |
```

Structural.34  Declaration Prologue

```
dec prol | n
range # | coor #

| .-------|unit or dec list 1 |
| . . .   |
| .-------|unit or dec list n |
```

Structural.35  Single Declaration List

```
s. d. l. | n
range # | coor #

| .-------|single declaration 1 |
| . . .   |
| .-------|single declaration n |
```
Structural.36  Tag List

```
tag list | n
range # | coor #
tag acc #1  (--> decblk 1)
...
tag acc #n  (--> decblk n)
```

Structural.37  Trimscript List

```
t.sc.list | n
range # | coor #

--------->"trimscript" 1
...

--------->"trimscript" n
```

Structural.38  Trimment List

```
trim. list | n
range # | coor #

--------->"trimment" 1
...

--------->"trimment" n
```
Structural Parade

<table>
<thead>
<tr>
<th>parade</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>train 1</td>
<td></td>
</tr>
<tr>
<td>train n</td>
<td></td>
</tr>
</tbody>
</table>

Declaration Mode Dec Type

<table>
<thead>
<tr>
<th>mode dec</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ind link</td>
<td></td>
</tr>
<tr>
<td>xreflink</td>
<td></td>
</tr>
<tr>
<td>mode dec acc #</td>
<td></td>
</tr>
<tr>
<td>mode dec value</td>
<td></td>
</tr>
<tr>
<td>mode dec value</td>
<td></td>
</tr>
</tbody>
</table>
### Declaration.2  Tag Dec Type

<table>
<thead>
<tr>
<th>tag dec</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>--&gt; indlink</td>
</tr>
<tr>
<td></td>
<td>--&gt; xreflink</td>
</tr>
<tr>
<td></td>
<td>--&gt; tag dec mode</td>
</tr>
<tr>
<td>tag dec acc #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--&gt; tag dec value</td>
</tr>
<tr>
<td>heap vs. loc</td>
<td></td>
</tr>
</tbody>
</table>

### Declaration.3  Op Dec Type

<table>
<thead>
<tr>
<th>op dec</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>--&gt; indlink</td>
</tr>
<tr>
<td></td>
<td>--&gt; xreflink</td>
</tr>
<tr>
<td>op dec acc #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--&gt; op dec l mode</td>
</tr>
<tr>
<td></td>
<td>--&gt; op dec r mode</td>
</tr>
<tr>
<td></td>
<td>--&gt; op dec d mode</td>
</tr>
<tr>
<td></td>
<td>--&gt; op dec value</td>
</tr>
</tbody>
</table>
### Declaration

<table>
<thead>
<tr>
<th>prio dec</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
</tbody>
</table>

- \(\rightarrow\) indlink
- \(\rightarrow\) xreflink

### Prio Dec Type


---

### Format 1

**Collection List**

<table>
<thead>
<tr>
<th>col. list</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
</tbody>
</table>

- \(\rightarrow\) "collection" 1
- \(\rightarrow\) "collection" n

---

### Format 2

**Collection**

<table>
<thead>
<tr>
<th>collection</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
</tbody>
</table>

- \(\rightarrow\) insertion or nil
- \(\rightarrow\) replicas or nil
- \(\rightarrow\) collection list or nil or collection
- \(\rightarrow\) insertion or nil
Format.3 Integral Pattern

<table>
<thead>
<tr>
<th>int patt</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
</tbody>
</table>

- \(\rightarrow\) sign mould
- \(\rightarrow\) integral mould

Format.4 Integral Mould

<table>
<thead>
<tr>
<th>int mould</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
</tbody>
</table>

- \(\rightarrow\) simple int mould
- \(\rightarrow\) insertion

Format.5 Integral Insertion Pattern

<table>
<thead>
<tr>
<th>int i pat</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
</tbody>
</table>

- \(\rightarrow\) int choice patt
- \(\rightarrow\) insertion
Format.6  Integral Choice Pattern

<table>
<thead>
<tr>
<th>int ch pat</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format.7  Sign Mould

<table>
<thead>
<tr>
<th>sign mld</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format.8  Loose Replicatable Zero Frame

<table>
<thead>
<tr>
<th>l r z frm</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Format.9   Simple Integral Mould

<table>
<thead>
<tr>
<th>s int mld</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

\[ \ldots \rightarrow \text{l r s d frame } 1 \]

\[ \ldots \rightarrow \text{l r s d frame } n \]

Format.10   Loose Replicatable Suppressible Digit Frame

<table>
<thead>
<tr>
<th>l r s d f</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

\[ \ldots \rightarrow \text{insertion} \]

\[ \ldots \rightarrow \text{r s d frame} \]

Format.11   Replicatable Suppressible Digit Frame

<table>
<thead>
<tr>
<th>rep s d f</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

\[ \ldots \rightarrow \text{replication} \]

\[ \ldots \rightarrow \text{sup r frame} \]
Format. 12  Suppressible Digit Frame

<table>
<thead>
<tr>
<th>sup d fr</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>+----------&gt; digit frame</td>
</tr>
</tbody>
</table>

Format. 13  Digit Frame

<table>
<thead>
<tr>
<th>dig frame</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td>letter z,u,v,d</td>
<td></td>
</tr>
</tbody>
</table>

Format. 14  Sign Frame

<table>
<thead>
<tr>
<th>sign fr</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td>+ or -</td>
<td></td>
</tr>
</tbody>
</table>
Format. 15  Literal List

<table>
<thead>
<tr>
<th>lit list</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td>• --- ---</td>
<td>&gt; &gt; literal 1</td>
</tr>
<tr>
<td>• • •</td>
<td></td>
</tr>
<tr>
<td>• --- ---</td>
<td>&gt; &gt; literal n</td>
</tr>
</tbody>
</table>

Format. 16  Literal

<table>
<thead>
<tr>
<th>literal</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td>• --- ---</td>
<td>&gt; &gt; string denotation</td>
</tr>
<tr>
<td>• --- ---</td>
<td>&gt; &gt; rep lit sequence</td>
</tr>
</tbody>
</table>

Format. 17  Replicated Literal Sequence

<table>
<thead>
<tr>
<th>r lit seq</th>
<th>2n</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td>• --- ---</td>
<td>&gt; &gt; replication 1</td>
</tr>
<tr>
<td>• --- ---</td>
<td>&gt; &gt; string deno 1</td>
</tr>
<tr>
<td>• • •</td>
<td></td>
</tr>
<tr>
<td>• --- ---</td>
<td>&gt; &gt; replication n</td>
</tr>
<tr>
<td>• --- ---</td>
<td>&gt; &gt; string deno n</td>
</tr>
</tbody>
</table>
Format.18  Bits Pattern

<table>
<thead>
<tr>
<th>bits pat</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format.19  Radix Mould

<table>
<thead>
<tr>
<th>radix mld</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format.20  Real Pattern

<table>
<thead>
<tr>
<th>real pat</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Format.21 Real Mould

<p>| real mould | 2 |</p>
<table>
<thead>
<tr>
<th>range #</th>
<th>coor #</th>
</tr>
</thead>
<tbody>
<tr>
<td>•---------------------</td>
<td>&gt; integral mould</td>
</tr>
<tr>
<td>•---------------------</td>
<td>&gt; integral mould</td>
</tr>
</tbody>
</table>

Format.22 Real Fractional Mould

<p>| r frac mld | 2 |</p>
<table>
<thead>
<tr>
<th>range #</th>
<th>coor #</th>
</tr>
</thead>
<tbody>
<tr>
<td>•---------------------</td>
<td>&gt; insertion</td>
</tr>
<tr>
<td>•---------------------</td>
<td>&gt; integral mould</td>
</tr>
</tbody>
</table>

Format.23 Floating Point Mould

<p>| fl pt mld | 3 |</p>
<table>
<thead>
<tr>
<th>range #</th>
<th>coor #</th>
</tr>
</thead>
<tbody>
<tr>
<td>•---------------------</td>
<td>&gt; stagnant mould</td>
</tr>
<tr>
<td>•---------------------</td>
<td>&gt; sign mould</td>
</tr>
<tr>
<td>•---------------------</td>
<td>&gt; integral mould</td>
</tr>
</tbody>
</table>
Format 24  Stagnant Mould

\[
\begin{array}{|c|}
\hline
\text{stag mld} & 2 \\
\hline
\text{range #} & \text{coor #} \\
\hline
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{• --- --- --- ---} \\
\rightarrow \text{sign mould} \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{• --- --- --- ---} \\
\rightarrow \text{real/integral mould} \\
\end{array}
\]

Format 25  Boolean Pattern

\[
\begin{array}{|c|}
\hline
\text{bool mould} & 2 \\
\hline
\text{range #} & \text{coor #} \\
\hline
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{• --- --- --- ---} \\
\rightarrow \text{simple bool pat} \\
\end{array}
\]

\[
\begin{array}{c}
\text{• --- --- --- ---} \\
\rightarrow \text{insertion} \\
\end{array}
\]

Format 26  Simple Boolean Pattern

\[
\begin{array}{|c|}
\hline
\text{s b pat} & 2 \\
\hline
\text{range #} & \text{coor #} \\
\hline
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{• --- --- --- ---} \\
\rightarrow \text{insertion or nil} \\
\end{array}
\]

\[
\begin{array}{c}
\text{• --- --- --- ---} \\
\rightarrow \text{boolean choice mld} \\
\end{array}
\]
Format.27  Boolean Choice Mould

<table>
<thead>
<tr>
<th>b ch mld</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>--&gt; true literal</td>
</tr>
<tr>
<td></td>
<td>--&gt; false literal</td>
</tr>
</tbody>
</table>

Format.28  Complex Pattern

<table>
<thead>
<tr>
<th>complex p</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>--&gt; real pattern</td>
</tr>
<tr>
<td></td>
<td>--&gt; real pattern</td>
</tr>
</tbody>
</table>

Format.29  String Pattern

<table>
<thead>
<tr>
<th>str pat</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>range #</td>
<td>coor #</td>
</tr>
<tr>
<td></td>
<td>--&gt; pattern or frame</td>
</tr>
<tr>
<td></td>
<td>--&gt; insertion</td>
</tr>
</tbody>
</table>
Format 30  Simple String Pattern

```
| spl str p | n |
|-----------|
| range #   | coor # |

  \rightarrow l r s c fr 1

\[ \cdots \]

  \rightarrow l r s c fr n
```

Format 31  Loose Replicatable Suppressible Character Frame

```
| l r s c fr | 2 |
|------------|
| range #    | coor # |

  \rightarrow insertion

\[ \cdots \]

  \rightarrow rep s ch frame
```

Format 32  Replicatable Suppressible Character Frame

```
| r s ch fr  | 2 |
|------------|
| range #    | coor # |

  \rightarrow replication

\[ \cdots \]

  \rightarrow supp ch frame
Format.33  Loose String Frame

| l str fr | 1 |
| range # | coor # |

Format.34  Loose General Frame

| l gen fr | 1 |
| range # | coor # |

Format.35  Insertion

| insertion | 2 |
| range # | coor # |

Literal

Insert sequence
Format.36   Insert Sequence

ins seq | n

<table>
<thead>
<tr>
<th>range #</th>
<th>coor #</th>
</tr>
</thead>
</table>

• • •

• • •

Format.37   Insert

insert | 3

<table>
<thead>
<tr>
<th>range #</th>
<th>coor #</th>
</tr>
</thead>
</table>

• • •

• • •

Format.38   Alignment

align | 1

<table>
<thead>
<tr>
<th>range #</th>
<th>coor #</th>
</tr>
</thead>
</table>

letter k,x,y,l,p