STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

PENULTIMATE DRAFT REPORT ON THE ALGORITHMIC LANGUAGE ALGOL 68

A. van Wijngaarden (Editor),
B.J. Mailloux, J.E.L. Peck
and C.H.A. Koster.

MR 99
October 1968

1. Language and metalanguage

1.1. The method of description

1.1.1 The strict, extended and representation languages

a) ALGOL 68 is a language in which "programs" can be formulated for "computers", i.e. "automata" or "human beings". It is defined in three stages, the "strict language", the "extended language" and "representation language".

b) For the definition partly the "English language", and partly a "formal language" is used. In both languages, and also in the strict language and the extended language, typographical or syntactic marks are used which bear no relations to those used in the representation language.

1.1.2. The Syntax of the strict language.

a) The strict language is defined by means of a syntax and semantics. This syntax is a set of "production rules" for "notions"; it is defined by means of "small syntactic marks", in this Report "abcdefghijklmnopqrstuvwxyz", "large syntactic marks", in this Report "ABCDEFGHIJKLMNOPQRSTUVWXYZ", and "other syntactic marks", in this Report, "point" ("."), "comma" (","), "colon" (":"), "semicolon" (";") and "asterisk" ("*"). {note that these marks are in another type font than the marks of this sentence.}

b) A "protonotion" is a nonempty, possibly infinite, sequence of small syntactic marks; a notion is a protonotion for which there is a production rule and a "symbol" is a protonotion ending with 'symbol'.

c) A production rule for a notion consists of that notion, possibly preceded by an asterisk, followed by a colon, followed by a "direct production" of that notion, i.e. a "list of notions", and followed by a point.

d) A list of notions is a nonempty sequence of "members" separated by commas; a member is either a notion and is then said to be "productive" {, or nonterminal,} or is a symbol {, which is terminal,} or is empty.

1.1.2. continued

e) A "production" of a given notion is either a direct production of that given notion or a list of notions obtained by replacing a productive member in some production of the given notion by a direct production of that productive member.

f) A "terminal production" of a notion is a production of that notion none of whose members is productive.

{In the production rule
'variable-point numeral : integral part option, fractional part.'
(5.1.2.1.b) of the strict language, the list of notions 'integral part option, fractional part' is a direct production of the notion 'variable point numeral', containing two members, both of which are productive. A terminal production of this same notion is
'digit zero symbol, point symbol, digit one symbol'.
The member, 'digit zero symbol', is an example of a (terminal) symbol.
The ~~line~~ protonotion 'twas brillig and the slithy toves' ~~is a protonotion but~~ is neither a symbol nor a notion in the sense of this Report, in that it does not end with 'symbol' and no production rule for it is given (1.1.5.b,c)}

1.1.3. The syntax of the metalanguage

a) The production rules of the strict language are partly enumerated and partly generated with the aid of a "metalanguage" whose syntax is a set of production rules for "metanotions".

b) A metanotion is a nonempty sequence of large syntactic marks.

c) A production rule for a metanotion consists of that metanotion followed by a colon, followed by a direct production of that metanotion, i.e., a "list of metanotions", and followed by a point.

d) A list of metanotions is a ~~possibly empty~~ nonempty sequence of "metamembers" separated by blanks; a metamember is either a metanotion and is then said to be productive, or is a ~~nonempty~~ , possibly empty, sequence of small syntactic marks.

1.1.3. continued

e) A production of a given metanotion is either a direct production of that given metanotion or a list of metanotions obtained by replacing a productive metamember in some production of the given metanotion by a direct production of that productive metamember.

f) A terminal production of a metanotion is a production of that metanotion none of whose metamembers is productive.

{In the production rule 'TAG : LETTER.', derived from 1.2.1.t, 'LETTER' is a direct production of the metanotion 'TAG', consisting of one metamember which is productive. A particular terminal production of the metanotion 'TAG' is 'letter x' (see 1.2.1.s,t). In the production rule 'EMPTY : .' (1.2.1.i), the metanotion 'EMPTY' has a direct production which consists of one empty metamember. }

1.1.4. The production rules of the metalanguage

The production rules of the metalanguage are the rules obtained from the rules in Section 1.2 in the following steps:

Step 1: If some rule contains one or more semicolons, then it is replaced by two new rules, the first one of which consists of the part of that rule up to and including the first semicolon with that semicolon replaced by a point, and the second of which consists of a copy of that part of the rule up to and including the colon, followed by the part of the original rule following its first semicolon, whereupon Step 1 is taken again;

Step 2: A number of production rules for the metanotion 'ALPHA' {1.2.1.t}, each of whose direct productions is another small syntactic mark, may be added.

{For instance, the rule 'TAG : LETTER ; TAG LETTER ; TAG DIGIT', from 1.2.1.t. is replaced by the rules 'TAG : LETTER.' and 'TAG : TAG LETTER ; TAG DIGIT.', and the second of these is replaced by 'TAG : TAG LETTER.' and 'TAG : TAG DIGIT.',
thus resulting in three rules from the original one.

The reader may find it helpful to read ":" as "may be a", "," as "followed by a", and ";" as "or a". }

1.1.5. The production rules of the strict language

a) The production rules of the strict language are all the rules obtained in the following steps from the rules given in Chapters 2 up to 8 inclusive under Syntax:

Step 1: Identical with Step 1 of 1.1.4 ;

Step 2: If the given rule now contains one or more metanotions, then for some terminal production of such a metanotion, a new rule is obtained by replacing that metanotion, throughout a copy of the given rule, by that terminal production, whereupon the given rule is discarded and Step 2 is taken; otherwise, all blanks in the given rule are removed and the rule so obtained is a production rule of the strict language.

b) A number of production rules may be added for the notion 'indicant' {4.2.1.b,e,f} each of whose direct productions is a symbol different from any symbol given in this Report { ; see also 3.1.2.c}.

c) A number of production rules may be added for the notions 'other comment item' {3.0.9.c} and 'other string item' {5.3.1.c} each of whose direct productions is a symbol different from any character-token with the restrictions that no other-comment-item is the comment-symbol and no other-string-item is the quote-symbol.

{The rule
'actual LOWPER bound : strict LOWPER bound.'
derived from 7.1.1.s by Step 1 is used in Step 2 to provide two production rules of the strict language, viz.
'actual lowerbound:strictlowerbound.' and
'actualupperbound:strictupperbound.' ;
however, to ease the burden on the reader, who may more easily ignore blanks himself, some blanks will be retained in the symbols, notions and production rules in the rest of this Report. Thus, the rules will be written in the more readable form
'actual lower bound : strict lower bound.' and
'actual upper bound : strict upper bound.'.

1.1.5. continued

Note that

'actual lower bound : strict upper bound.'

is not a production rule of the strict language, since the replacement of the metanotion 'LOWPER' by one of its productions must be consistent throughout. Since some metanotions have an infinite number of terminal productions, the number of notions of the strict language is infinite and the number of production rules for a given notion may be infinite; moreover, since some metanotions have terminal productions of infinite length, some notions are infinitely long. For examples see 4.1.1 and 8.5.2.2. Some production rules obtained from a rule containing a metanotion may be blind alleys in the sense that no production rule is given for some member to the right of the colon even though it is not a symbol. }

1.1.6. The semantics of the strict language

a) A terminal production of a notion is considered as a linearly ordered sequence of symbols. This order is called the "textual order", and "following" ("preceding") stands for "textually immediately following" ("textually immediately preceding") in the rest of this Report. Typographical display features, such as blank space, change to a new line, and change to a new page do not influence this order.

b) A sequence of symbols (A protonotion) consisting of a second sequence of symbols (a second protonotion) preceded and/or followed by (a) nonempty sequence(s) of symbols (of small syntactic marks) "contains" that second sequence of symbols (second protonotion).

c) A "paranotion" when not in a section whose heading is or begins with ~~under~~ "Syntax", not between "apostrophes" (" ' ") and not within another paranotion "denotes" some number of protonotions.

A paranotion is

i) a symbol and it then denotes itself {e.g., "begin symbol" denotes "begin symbol" }, or

ii) a notion whose production rule(s) do(es) not begin with an asterisk, and it then denotes itself {,e.g., "plusminus" denotes "plusminus"}, or

1.1.6. continued

iii) a notion whose production rule(s) do(es) begin with an asterisk, and it then denotes any of its direct productions {, which, in this Report, always is a notion or a symbol, e.g., "trimscript" (8.6.1.1.j) denotes "trimmer option" or "subscript"}, or

iv) a paranotion in which one or more "hyphen"s ("-") have been inserted and it then denotes those protonotions denoted by that paranotion before the insertion(s) {, e.g., "begin-symbol" denotes what "begin symbol" denotes}, or

v) a paranotion followed by "s" or a paranotion ending with "y" in which that "y" has been replaced by "ies" and it then denotes some number of those protonotions denoted by that paranotion before the modifications {, e.g., "trimscripts" denotes some number of "trimmer option"s and/or "subscript"s and "primaries" denotes some number of the notions denoted by "primary"}, or

vi) a paranotion whose first small syntactic mark has been replaced by the corresponding large syntactic mark, and it then denotes those protonotions denoted by that paranotion before the modification {, e.g., "Identifiers" denotes the notions denoted by "identifiers"}, or

vii) a paranotion in which a terminal production of 'SORT' and/or of 'SOME' and/or of 'MOID' has been omitted, and it then denotes those protonotions denoted by any other paranotion from which the given paranotion could be obtained by omitting a terminal production of 'SORT' and/or of 'SOME' and/or of 'MOID' ⊥ {, e.g., "hop" denotes the notions denoted by "MOID hop" (8.2.7.1.b), "declaration" denotes the notions denoted by "SOME declaration" (6.2.1.a, 7.0.1.a) and "clause" denotes the notions denoted by "SORTETY SOME MOID clause" (6.0.1.a, 6.0.2.b,c,d,f, 6.3.1.a, 6.4.1.a,c,d,e, 8.1.1.a), where "SORTETY" ("SOME", "MOID") stands for any terminal production of the metanotion 'SORTETY' ('SOME', 'MOID')}.

{As an aid to the reader, paranotions, when not under Syntax or between apostrophes, are provided with hyphens where, otherwise, they are provided with blanks. Rules beginning with an asterisk have been included in order to shorten the semantics. }

⊥, and if that other paranotion is a notion, then it is said to "generate" the given paranotion

d) Except as otherwise specified {f, g} , a paranotion stands for any occurrence of any symbol denoted by it and/or of any terminal production of any notion denoted by it.

e) An occurrence of a protonotion which is a member of a direct production of a given occurrence of a notion is a "direct constituent" of that occurrence of that notion; an occurrence of a protonotion which is a member of a given production of a given occurrence of a notion is a "constituent" of that given occurrence of that notion, provided that it is not also a member of a production of another occurrence of either that notion or that protonotion which other occurrence is a member of the given production.

{The terminal production of 'integral-slice' (8.6.1.1.a) S1 , viz. i2[1,i1[1]], contains three occurrences of 'digit-one-symbol' (3.1.1.b), 1, two occurrences of 'sub-symbol' (3.1.1.e) , [, and one occurrence of a terminal production of 'integral-slice' S2, viz. i1[1], which is a constituent of S1. The first occurrence of 1 is a constituent of S1; the second and third are constituents of S2 and, since S2 is both 'integral-slice' and a constituent of S1, not constituents of S1. The first occurrence of [is a direct constituent of S1 and the second is a direct constituent of S2 but not a constituent of S1. }

f) A paranotion *denoting the occurrences of protonotions* of all of which are (direct) constituents of occurrences of notions denoted by a second paranotion is a (direct) constituent of that second paranotion. {e.g., since paranotions stand for occurrences of terminal productions (d), j := 1 is a constituent assignation (8.3.1.1.a) of the assignation i := j := 1, but not of the serial-clause (6.1.1.a) i := j := 1; k := 2 nor of the assignations j := 1 and k := i := j := 1. The assignation j := 1 is not a direct constituent of the assignation i := j := 1, but it is a direct constituent source of that assignation (8.3.1.1.b).}

g) A paranotion which is a direct constituent of a second paranotion is a paranotion of that second paranotion {; i.e., "direct constituent of", which would occur frequently under Semantics, will usually be shortened to "of", "its" or even "the", e.g., in i := 1, i is its destination (8.3.1.1.b,c) or i is the or a destination of i := 1, whereas, i is a constituent destination but not simply a destination of the serial-clause i := 1 ; j := 2.}.

h) In sections 2 up to 8 under "Semantics" a meaning is associated with occurrences of certain sequences of symbols by means of sentences in the English language, as a series of processes (the "elaboration" of those occurrences of sequences of symbols as terminal productions of given notions), each causing a specific effect. Any of these processes may be replaced by any process which causes the same effect.

i) If a sequence of symbols is a terminal production of a given notion and another notion which is a direct production of the given notion, then its "preelaboration" ("prevalue", "premode", "prescope") as terminal production of the given notion is its elaboration ("value", "mode", "scope") as terminal production of that other notion; except as otherwise specified {8.2}, elaboration (value, mode, scope) of a sequence of symbols as terminal productions of a given notion is its preelaboration (prevalue, premode, prescope) as terminal production of that notion. {e.g., the elaboration (value, mode, scope) of the reference-to-real-confrontation (8.3.0.1.a) x := 3.14 is its preelaboration which is its elaboration (value, mode, scope) as a reference-to-real-nonlocal-assignation.}

{The syntax of the strict language has been chosen in such a way that a given sequence of symbols which is a terminal production of 'program' is so by means of a unique set of productions, except, possibly, for production rules inducing ~~only~~ at most preelaboration, e.g. derived from rules 6.2.1.e and 6.4.1.d (balancing of modes; see also 2.3.a) and from rule 7.1.1.z in combination with 7.2.1.a and 7.4.1.a (order of terminal productions of 'MOOD' in a terminal production of 'UNITED'').}

k) If something is left undefined or is said to be undefined, this means that it is not defined by this Report alone, and that, for its definition, information from outside this Report has to be taken into account.

j) A terminal production of a given metanotion is "enveloped" by a given notion if it is contained once in that notion but not in another terminal production of that metanotion contained in that notion {; e.g., 'reference to real' is enveloped as terminal production of 'MODE' by 'reference to real mode identifier', but 'real' is not}.

1.1.7. The extended language

The extended language encompasses the strict language; i.e., a program in the strict language, possibly subjected to a number of notational changes by virtue of "extensions" given in Chapter 9 is a program in the extended language and has the same meaning. {e.g., <u>real</u> x, y, z means the same as <u>real</u> x, <u>real</u> y, <u>real</u> z by 9.2.c.}

1.1.8. The representation language

a) The representation language represents the extended language; i.e., a program in the extended language, in which all symbols are replaced by certain typographical marks by virtue of "representations", given in section 3.1.1, and in which all commas {not comma-symbols} are deleted, is a program in the representation language and has the same meaning.

b) Each version of the language in which representations are used which are sufficiently close to the given representations to be recognised without further elucidation is also a representation language. A version of the language in which notations or representations are used which are not obviously associated with those defined here, is a "publication language" or "hardware language" {i.e. a version of the language suited to the supposed preference of the human or mechanical interpreter of the language}.

{e.g., <u>begin</u>, **begin**, and 'BEGIN' are all representations of the begin-symbol in the representation language.}

begin

begin

begin

begin

begin

begin

begin

1.2 The metaproduction rules

1.2.1. Metaproduction rules of modes

a) MODE : MOOD ; UNITED.
b) MOOD : TYPE ; STOWED.
c) TYPE : PLAIN ; format ; PROCEDURE ; reference to MODE.
d) PLAIN : INTREAL ; boolean ; character.
e) INTREAL : INTEGER ; REAL.
f) INTEGRAL : LONGSETY integral.
g) REAL : LONGSETY real.
h) LONGSETY : long LONGSETY ; EMPTY.
i) EMPTY : .
j) PROCEDURE : procedure PARAMETY MOID.
k) PARAMETY : with PARAMETERS ; EMPTY.
l) PARAMETERS : PARAMETER ; PARAMETERS and PARAMETER.
m) PARAMETER : MODE parameter.
n) MOID : MODE ; void.
o) STOWED : structured with FIELDS ; row of MODE.
p) FIELDS : FIELD ; FIELDS and FIELD.
q) FIELD : MODE field TAG.
r) TAG : LETTER ; TAG LETTER ; TAG DIGIT.
s) LETTER : letter ALPHA.
t) ALPHA : a ; b ; c ; d ; e ; f ; g ; h ; i ; j ; k ; l ; m ; n ; o ;
p ; q ; r ; s ; t ; u ; v ; w ; x ; y ; z ; aleph.
u) DIGIT : digit FIGURE.
v) FIGURE : zero ; one ; two ; three ; four ; five ; six ; seven ;
eight ; nine.
w) UNITED : union of MOOD and MOODS mode ;
x) MOODS ; MOOD ; MOODS and MOOD.

{The reader may find it helpful to note that a metanotion ending in 'ETY' always has 'EMPTY' as a direct production.}

1.2.2. Metaproduction rules associated with modes

a) PRIMITIVE : integral ; real ; boolean ; character ; format.
b) ROWS : row of ; ROWS row of.
c) ROWSETY : ROWS ; EMPTY.

1.2.2. continued

d) ROWWSETY : ROWSETY.

e) NONROW : NONSTOWED ; structured with FIELDS.

f) NONSTOWED : TYPE ; UNITED.

g) REFETY : reference to ; EMPTY.

h) NONPROC : PLAIN ; format ; procedure with PARAMETERS MOID ; reference to NONPROC ; structured with FIELDS ; row of NONPROC ; UNITED.

i) PRAM : procedure with LMODE parameter and RMODE parameter MOID ; procedure with RMODE parameter MOID .

j) LMODE : MODE.

k) RMODE : MODE.

l) MOOT : MOOD.

m) LMOODSETY : MOODS and ; EMPTY.

n) RMOODSETY : and MOODS ; EMPTY.

o) LOSETY : LMOODSETY.

p) BOX : LMOODSETY box.

q) LFIELDSETY : FIELDS and ; EMPTY.

r) RFIELDSETY : and FIELDS ; EMPTY.

s) COMPLEX : structured with real field letter r letter e and real field letter i letter m.

t) BITS : structured with row of boolean field LENGTHETY letter aleph.

u) LENGTHETY : LENGTH LENGTHETY ; EMPTY.

v) LENGTH : letter l letter o letter n letter g.

w) BYTES : structured with row of character field LENGTHETY letter aleph.

x) STRING : row of character ; character.

y) MABEL : MODE mode ; label.

1.2.3. Metaproduction rules associated with phrases and coercion

a) PHRASE : declaration ; CLAUSE.
b) CLAUSE : MOID clause.
c) SOME : serial ; unitary ; CLOSED ; choice ; THELSE.
d) CLOSED : closed ; collateral ; conditional.
e) THELSE : then ; else.
f) SORTETY : SORT ; EMPTY.
g) SORT : strong ; FIRM.
h) FEAT : firm ; weak ; soft.
i) STRONGETY : strong ; EMPTY.
j) STIRM : strong ; firm.
k) ADAPTED : ADJUSTED ; widened ; rowed ; hipped ; voided.
l) ADJUSTED : FITTED ; procedured ; united.
m) FITTED : dereferenced ; deprocedured.

1.2.4. Metaproduction rules associated with coercends

a) COERCEND : MOID FORM.
b) FORM : confrontation ; FORESE.
c) FORESE : ADIC formula ; cohesion ; base.
d) ADIC : PRIORITY ; monadic.
e) PRIORITY : priority NUMBER.
f) NUMBER : one ; TWO ; THREE ; FOUR ; FIVE ; SIX ; SEVEN ;
   EIGHT ; NINE.
g) TWO : one plus one.
h) THREE : TWO plus one.
i) FOUR : THREE plus one.
j) FIVE : FOUR plus one.
k) SIX : FIVE plus one.
l) SEVEN : SIX plus one.
m) EIGHT : SEVEN plus one.
n) NINE : EIGHT plus one.

1.2.5. Other metaproduction rules

a) VICTAL : VIRACT ; formal.

b) VIRACT : virtual ; actual.

c) LOWPER : lower ; upper. d) LOCAL : local; nonlocal.

e) ANY : KIND ; suppressible KIND ; replicatable KIND ; replicatable suppressible KIND.

f) KIND : sign ; zero ; digit ; point ; exponent; complex ; string ; character.

g) NOTION : ALPHA ; NOTION ALPHA.

h) SEPARATOR : LIST separator ; go on symbol ; completer ; sequencer ; statement interlude option.

i) LIST : list ; sequence.

{Rule g implies that all protonotions (1.1.2.b) are productions (1.1.3.e) of the metanotion (1.1.3.b) 'NOTION' ; for the use of this metanotion, see 3.0.1.b,c,d,g,h.}

{"Well 'slithy' means 'lithe' and 'slimy'. ...
You see it's like a portmanteau - there are
two meanings packed into one word."
Through the Looking Glass, Lewis Carroll. }

1.3. Pragmatics

{Merely corroborative detail, intended to
give artistic verisimilitude to an other-
wise bald and unconvincing narrative.
Mikado. W.S. Gilbert. }

Scattered throughout this Report are "pragmatic" remarks included between the braces { and }. These do not form part of the definition of the language but are intended to help the reader to understand the implications of the definitions and to find corresponding sections or rules.

{The rules under Syntax are interpreted with cross-references to be interpreted as follows. Let a "hypernotion" be either a protonotion or a sequence of one or more metanotions, possibly preceded and/or separated and/or followed by protonotions; then, each rule consists of a hypernotion followed by a colon followed by one or more hypernotions separated by commas or semicolons, and is closed by a point. By virtue of 1.1.5.a. Step 2, each hypernotion yields eventually one or more protonotions.

1.3. continued

In each rule, a hypernotion appearing before (after) the colon is followed by indicators of the rules in which a hypernotion yielding one or more protonotions also yielded by the first hypernotion appears after (before) the colon or indicators of the representations in section 3.1.1 of the symbols yielded by the first hypernotion. Here, an indicator is, in principle, the section number followed by the letter indicating the line where the rule or representation appears, with the following conventions:

i) the indicators whose section number is that of the section in which they appear, are given first and their section number is omitted; e.g., "3.0.3.b" appears as "b" in section "3.0.3";

ii) all points are omitted and 10 appears as A; e.g., "3.0.3.a" appears as "303a" elsewhere;

iii) a final 1 is omitted; e.g., "811a" appears as "81a";

iv) a section number which is the same as in the preceding indicator is omitted; e.g., "821a,821b" appears as "821a,b";

v) numerous indicators of the rules 3.0.1.b up to h are replaced by more helpful indicators; e.g., in 6.1.1.d, "chain of strong void units separated by go on symbols {30c}" appears as "chain of strong void units {e} separated by go on symbols {31f}"; also, indicators in section 3.0.1 are restricted to a bare minimum;

vi) the absence of a production rule for one or more protonotions which are not symbols and are yielded by a hypernotion appearing after that colon, is indicated by "-"; e.g., in 8.6.0.1.a after "MOID$\overset{mode}{\wedge}$identifier" appears {41b,-} since 4.1.1.b yields production rules for all notions yielded by 'MODE$\overset{mode}{\wedge}$identifier' but not for 'void$\overset{mode}{\wedge}$identifier', and no other rule does.

Some of the pragmatic remarks are examples in the representation language. In these examples, identifiers occur out of context from their defining occurrences. Unless otherwise specified, these occurrences identify those in the standard-prelude (e.g., see 10.3.k for random and 10.3.a for pi), or those in:

```
int i, j, k, m, n ; real a, b, x, y ; bool p, q, overflow ;
char c ; format f ; bytes r ; string s ; bits t ; compl w, z ;
ref real xx, yy ; [1:n]real x1, y1 ; [1:m,1:n]real x2 ;
[1:n,1:n]real y2 ; [1:n]int i1 ; [1:m,1:n]int i2 ;
proc xory = ref real : (random < .5 | x | y) ;
proc ncos = (int i)real : cos(2 × pi × i / n) ;
proc nsin = (int i)real : sin(2 × pi × i / n) ;
proc g = (real u)real : (arctan(u) - a + u - 1) ;
proc stop = (:(ℓ ; ℓ));
exit : princeton : grenoble : st pierre de chartreuse :
kootwijk : warsaw : zandvoort : amsterdam : tirrenia :
north berwick : x := 1. }
```

2. The computer and the program

{The programmer is concerned with particular-programs (2.1.d). These are always contained in a program (2.1.a), which also contains a standard-prelude, i.e. a declaration-prelude which is always the same (see Chapter 10), and possibly a library-prelude, i.e. a declaration-prelude which may depend upon the implementation.}

2.1. Syntax

a) program : open symbol{31e}, standard prelude{b},
   library prelude{c} option, particular program{d},
   close symbol{31e}.

b) standard prelude{a} : declaration prelude{61b}.

c) library prelude{a} : declaration prelude{61b}.

d) particular program{a} : label{61k} sequence option,
   open symbol{31e}, strong serial void clause{61a}, close symbol{31e}.

2.2. Terminology

{"When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean - neither more nor less."
Through the Looking Glass, Lewis Carroll. }

The meaning of a program is explained in terms of a hypothetical computer which performs a set of "actions" {2.2.5}, the elaboration of the program {2.3.a}. The computer deals with a set of "objects" {2.2.1} between which, at any given time, certain "relationships" {2.2.2} may "hold".

2.2.1 Objects

Each object is either "external" or "internal". External objects are occurrences of terminal productions {1.1.2.f} of notions. Internal objects are "instances" of "values" {2.2.3}.

2.2.2. Relationships

a) Relationships either are "permanent", i.e. independent of the program and its elaboration, or actions may cause them to hold or cease to hold.

2.2.2. continued

Each relationship is either between external objects or between an external object and an internal object or between internal objects.

b) The relationships between external objects are: to contain {1.1.6.b}, to be a "constituent" or "direct constituent" of {1.1.6.e} and "to identify" {c}.

c) A given occurrence of a terminal production of 'MABEL identifier' {4.1.a} ('MODE mode indication'{4.2.1.b} or 'PRIORITY indication'{4.2.1.e}, 'PRAM ADIC operator'{4.3.1.c,d}) where "MABEL" ("MODE", "PRIORITY", "PRAM", "ADIC") stands for any terminal production of the metanotion 'MABEL' ('MODE', 'PRIORITY', 'PRAM', 'ADIC') may identify a "defining occurrence"1) of the same terminal production.

1 ("indication-defining occurrence", "operator-defining occurrence")

d) The relationship between an external object and an internal object is: "to possess".

e) An external object considered as an occurrence of a terminal production of a given notion may possess an instance of a value, termed "the" value of the external object when it is clear which notion is meant; in general, "an (the) instance of a (the) value" is sometimes shortened in the sequel to "a (the) value" when it is clear which instance is meant.

f) A mode-identifier (operator) may possess a value ({more specifically} a "routine" {2.2.3.4}). This relationship is caused to hold by the elaboration of an identity-declaration {7.4.1.a} (operation-declaration {7.5.1.a}) and ceases to hold upon the end of the elaboration of the smallest serial-clause {6.1.1.a} containing that declaration.

g) An external object other than an identifier or operator {e.g. serial-clause (6.1.1.a)} considered as a terminal production of a given notion may be caused to possess a value by its elaboration as terminal production of that notion, and continues to possess that value until the next elaboration, if any, of ~~the same occurrence of~~ that external object is "initiated", whereupon it ceases to possess that value.

h) The relationships between internal objects ~~values~~ are: "to be of the same mode as", "to be equivalent to", "to be smaller than", "to be a component of" and "to refer to". A relationship said to hold between a given value and a (an instance of a) second value holds between any instance of the given value and any (that) instance of the second value.

i) ~~A~~ An instance of a value may be of the same mode as another one; this relationship is permanent {2.2.4.1.a}.

j) A value may be equivalent to another value {2.2.3.1.d,f} and a value may be smaller than another value {2.2.3.1.c}. If one of these relationships is defined at all for a given pair of values, then either it does not hold, or it does hold and is permanent.

k) ~~A~~ An instance of a given value is a component of another one if it is a "field" {2.2.3.2} , "element" {2.2.3.3.a} or "subvalue" {2.2.3.3.c} of that other one or of one of its components.

l) Any "name" {2.2.3.5}, except "nil" {2.2.3.5.a}, refers to one instance of another value. This relationship {may be caused to hold by an "assignment" (8.3.1.2.c) of that instance of that value to that name and} continues to hold until another instance of a value is caused to be referred to by that name. The words "refers to an instance of" are often shortened in the sequel to "refers to".

2.2.3. Values

Values are

~~i)~~ "plain values"{2.2.3.1}, ~~which are independent of the program and its elaboration;~~

~~ii)~~ "structured" values {2.2.3.2} or "multiple" values {2.2.3.3}, which are composed of other values, ~~in a way defined by the program;~~

~~iii)~~ "routines" and "formats" {2.2.3.4}, which are certain sequences of symbols ~~defined by the program, or~~ and

~~iv)~~ names ~~{2.2.3.5}, which are created by the elaboration of the program.~~ {2.2.2.l, 2.2.3.5}.

2.2.3.1. Plain values

a) A plain value is either an "arithmetic" value, i.e. an integer or a real number, or is a truth value or character.

b) An arithmetic value has a "length number", i.e. a positive integer characterising the degree of discrimination with which the value is kept in the computer. The number of integers (real numbers) of given length number that can be distinguished increases with the length number up to a certain length number, the number of different lenghts of integers (real numbers) {10.2.a,c}, after which it is constant.

c) For each pair of integers (real numbers) of the same length number, the relationship to be smaller than is defined {10.3.2.a, 10.3.3.a}. For each pair of integers of the same length number, a third integer of that length number may exist, the first integer "minus" the other one {10.3.2.g}. Finally, for each pair of real numbers of the same length number, three real numbers of that length number may exist, the first real number "minus" ("times", "divided by") the other one {10.3.3.g,l,m,}; these real numbers are obtained "in the sense of numerical analysis", i.e. by performing the operations known in mathematics by these terms on real numbers which may deviate slightly from the given ones {; this deviation is left undefined in this Report}.

d) Each integer of given length number is equivalent to a real number of that length number. Also, each integer (real number) of given length number is equivalent to an integer (real number) whose length number is greater by one. These equivalences permit the "widening" {8.2.5} of an integer into a real number and the increase of the length number of an integ~~er~~ or real number. The inverse transformations are only possible on those real numbers (arithmetic values) which are equivalent to an integer (a value of smaller length number}.

e) A truth value is either "true" or "false".

2.2.3.1. continued

f) Each character has an "integral equivalent" {10.1.h}, i.e. a non-negative integer of length number one; this relationship is defined only in so far that different characters have different integral equivalents.

2.2.3.2. Structured values

{Yea, from the table of my memory
I'll wipe away all trivial fond records.
Hamlet, William Shakespeare.}

A structured value is composed of a number of other values, its fields, in a given order, each of which is "selected" {8.5.2.2.Step 2} by a specific field-selector {7.1.1.i}.

2.2.3.3. Multiple values

a) A multiple value is composed of a "descriptor" and a number of other values, its elements, each of which is selected {8.6.1.2. Step 7} by a specific integer, its "index".

b) The descriptor consists of an "offset", c, and some number, $n \geq 0$, of "quintuples" $(l_i, u_i, d_i, s_i, t_i)$ of integers, $i = 1, \ldots, n$; l_i is the i-th "lower bound", u_i the i-th "upper bound", d_i the i-th "stride", s_i the i-th "lower state" and t_i the i-th "upper state". If for any i, $i = 1, \ldots, n$, $u_i < l_i$, then the number of elements in the multiple value is zero; otherwise, it is

$(u_1 - l_1 + 1) \times \ldots \times (u_n - l_n + 1)$.

The descriptor "describes" each element for which there exists an n-tuple $(r_1, \ldots, r_n)$ of integers satisfying, for each $i = 1, \ldots, n$, $l_i \leq r_i \leq u_i$, and that element is selected by $c + (r_1 - l_1) \times d_1 + \ldots + (r_n - l_n) \times d_n$.

{To the name referring to a given multiple value a state of which is 1, no multiple value can be assigned (8.3.1.2.c. Step 4) in which the bound corresponding to that state differs from that in the given value.}

c) A subvalue of a given multiple value is a multiple value which is (is referred to by) the value of a slice {8.6.1} the value of whose primary {8.6.1.1.a} is (refers to) the given multiple value.

2.2.3.4. Routines and formats

A routine (format) is a sequence of symbols which is the same as some closed-clause {6.3.1.a} (format-denotation {5.5.1.a}).

2.2.3.5. Names

a) There is one name, nil, whose "scope" {2.2.4.2} is the program and which does not refer to any value; any other name is created by the elaboration of an actual-declarer {7.1.2.c. Step 8}✓ and refers to precisely one instance of a value}.

, a rowed-coercend {8.2.6.2. Step 7} or a skip {8.2.7.2.a} {,

b) If a given name refers to a structured value {2.2.3.2} , then to each of its fields there refers a name uniquely determined by the given name and the field-selector selecting that field, and whose scope is that of the given name.

c) If a given name refers to a given multiple value {2.2.3.3}, then to each element (each multiple value composed of a descriptor and elements which are a proper subset of the elements) of the given multiple value there refers a name uniquely determined by the given name and the index of that element (and that descriptor and that subset), and whose scope is that of the given name.

2.2.4. Modes and scopes

2.2.4.1. Modes

a) A "mode" is any terminal production of 'MODE' {1.2.1.a}. Each instance {2.2.1} of a value is of one specific mode which is a terminal production of 'MOOD' {1.2.1.b}; furthermore, all instances of a given value other than nil {2.2.3.5.a} are of one same mode, the mode of that given value, and a "copy" of a given instance of a value is a new instance of that value which is of the same mode as the given instance.

b) The mode of a truth value (character, format) is 'boolean' ('character', 'format').

c) The mode of an integer (a real number) of length number n is (n - 1) times 'long' followed by 'integral' (by 'real').

2.2.4.1. continued

d) The mode of a structured value is 'structured with' followed by one or more "portrayals" separated by 'and', one corresponding to each field taken in the same order, each portrayal being a mode followed by 'field' followed by a terminal production of 'TAG' {1.2.1.l} whose terminal production {field-selector} selects {2.2.3.2} that field; it is "structured from" a second mode if the mode in one of its portrayals is, or is structured from it.

e) The mode of a multiple value is a terminal production of 'NONROW' {1.2.2.e} preceded by as many times "row of" as there are quintuples in the descriptor of that value.

f) The mode of a routine is a terminal production of 'PROCEDURE' {1.2.1.q}.

g) The mode of a name is 'reference to' followed by another mode. {See 7.1.2.Step 8.}.

2.2.4.2. Scopes

a) Each value has one specific scope.

b) The scope of a plain value is the program,
that of a structured (multiple) value is the smallest of the scopes of its fields (elements),
that of a routine or format possessed by a given denotation {5.4.1.a, 5.5.1.a} is the smallest range {4.1.1.e} containing a defining occurrence {4.1.2.a} (indication-defining occurrence {4.2.2.a}, operator-defining occurrence {4.3.2.a}) of a terminal production, if any, an applied occurrence of which but not a defining occurrence (indication-defining occurrence, operator-defining occurrence) of which is contained in that denotation, and otherwise, the program, and
that of a name is some {8.5.1.2.b} range.

2.2.5. Actions

{Suit the action to the word,
the word to the action.
Hamlet, William Shakespeare.}

An action is "elementary" or "collateral". A serial action consists of actions which take place one after the other.

A collateral action consists of actions merged in time; i.e., it consists of the elementary actions which make up those actions provided only that each elementary action of each of those actions which would take place before another elementary action of the same action when not merged with the other actions, also takes place before it when merged.

The elaboration of the closed-clause {6.3.1.a} following the first do-symbol {3.1.1.h} contained in the actual-parameter of the operation-declaration {7.5.1.a} 10.4.a and the elaboration of the actual-parameter of the operation-declaration 10.4.b are elementary actions.
{What other actions are elementary is left undefined.}

2.3. Semantics

{"I can explain all the poems that ever were invented, - and a good many that haven't been invented just yet."
Through the Looking Glass,
Lewis Carroll. }

a) The elaboration of a program is the elaboration of the closed-clause {6.3.1.a} consisting of the same sequence of symbols. {In this Report, the Syntax says which sequences of symbols are programs and the Semantics which actions are performed by the computer when elaborating a program. Both Syntax and Semantics are recursive. Though certain sequences of symbols may be terminal productions of 'program' in more than one way (1.1.6.i), this syntactic ambiguity does not lead to a semantic ambiguity.}

b) In ALGOL 68, a specific notation for external objects is used which, together with its recursive definition, makes it possible to handle and to distinguish between arbitrarily long sequences of symbols, to distinguish between arbitrarily many different values of a given mode (except 'boolean') and to distinguish between arbitrarily many modes, which allows arbitrarily many objects to occur in the computer and which allows the elaboration of a program to involve an arbitrarily large, not necessarily finite, number of actions. This is not meant to imply that the notation of the objects in the computer is that used in ALGOL 68 nor that it has the same possibilities. It is, on the contrary, not assumed that the computer can handle arbitrary amounts of presented information.

It is not assumed that these two notations are the same or even that a one-to-one correspondence exists between them; in fact, the set of different notations of objects of a given category may be finite. It is not assumed that the speed of the computer is sufficient to elaborate a given program within a prescribed lapse of time, nor that the number of objects and relationships that can be established is sufficient to elaborate it at all.

c) A model of the hypothetical computer, using a physical machine, is said to be an "implementation"of ALGOL 68, if it does not restrict the use of the language in other respects than those mentioned above. Furthermore, if a language is defined whose particular-programs are particular-programs of ALGOL 68 and have the same meaning, then that language is said to be a sublanguage of ALGOL 68. A model is said to be an implementation of a sublanguage if it does not restrict the use of the sublanguage in other respects than those mentioned above.

{A sequence of symbols which is not a program but can be turned into one by deleting or inserting a certain number of symbols and not a smaller number could be regarded as a program with that number of syntactical errors. Any program that can be obtained by deleting or inserting that number of symbols may be termed a "possibly intended" program. Whether a program or one of the possibly intended programs has the effect its author in fact intended it to have, is a matter which falls outside this Report. }

{In an implementation, the particular-program may be "compiled", i.e. translated into an "object program" in the code of the physical machine. Under circumstances, it may be advantageous to compile parts of the particular-program independently, e.g. parts which are common to several particular-programs. If such a part contains mode-identifiers (indications, operators) whose defining (indication-defining, operator-defining) occurrences (Chapter 4) are not contained in that part, then compilation into an efficient object program may be assured by preceding the part by a chain of formal-parameters (5.4.1.g) (mode-declarations (7.2.1.a) or priority-declarations (7.3.1.a), captions (7.5.1.b)) containing those defining (indication-defining, operator-defining) occurrences. }

{The definition of specific sublanguages and also the specification of actions not definable by any program (e.g., compilation or initiation of the elaboration), is not given in this Report.
However, the definition of the language allows, for instance, to let a special representation of the comment-symbol different from the ones given in 3.1.1.i, viz. **c** or **comment**, preferably **p** or **pragma**, have the effect that by a comment (3.0.9.b) beginning with and ending on this special representation, the computer is invited to implement some such sublanguage or ALGOL 68 itself or to take some such undefinable action, as may be specified by the comment (e.g., **p** algol 68 **p**, **p** run **p** or **p** dump **p**). }

{**p** algol 68 **p**
begin **proc** **p** nonrec **p**
    p = p ; p **end**
p run **p** **p** ? **p**
Report on the Algorithmic
Language ALGOL 68. }

3. Basic tokens and general constructions

3.0. Syntax

3.0.1. Introduction

a)* basic token : letter token{302a} ; denotation token{303a} ;
    action token{304a} ; declaration token{305a} ;
    syntactic token{306a} ; sequencing token{307a} ;
    hip token{308a} ; extra token{309a} ; special token{30Aa}.

b) NOTION option : NOTION ; EMPTY.

c) chain of NOTIONs separated by SEPARATORs {c,d} : NOTION ;
    NOTION, SEPARATOR {e,f,31f,61d,j,l} ,
    chain of NOTIONs separated by SEPARATORs {c}.

d) NOTION LIST : chain of NOTIONs separated by LIST separators
    {c,e,f}.

e) list separator : comma symbol{31e}.

f) sequence separator : EMPTY.

g) NOTION LIST proper : NOTION, LIST separator {e,f}, NOTION
    LIST {d}.

h) NOTION pack : open symbol{31e}, NOTION, close symbol{31e}.

{Examples:

a) a ; 0 ; + ; int ; if ; . ; nil ; for ; " ;

b) 0 ; ; c) 0, 1, 2 ;

d) 0 ; 0, 1, 2 ; e) , ;

f) ; g) 1, 2, 3 ; h) (1, 2, 3) }

3.0.2. Letter token

a)* letter token : LETTER {b}.

b) LETTER{309d,41b,c,d,55h,i,o,q,552b,e,f,553f,554a,555b,556b,
    557b,71j} : LETTER symbol{31a}.

{Examples:

a) a ; (see 1.1.4.Step 2) }

{Letter-tokens either are or are constituents of identifiers (4.1.1.a), field-selectors (7.1.1.i), format-denotations (5.5.1.a) and row-of-character-denotations (5.3.1.a). }

3.0.3. Denotation tokens

a)* denotation token : number token{b} ; true symbol{31b} ;
    false symbol{31b} ; formatter symbol{31b} ;
    routine symbol{31b} ; flipflop{e} ; space symbol{31b}.
b) number token{309d} : digit token{c} ; point symbol{31b} ;
    times ten to the power symbol {31b}.
c) digit token{b, 511b} : DIGIT{d}.
d) DIGIT{c, 41d, 511a, 552c} : DIGIT symbol{31b}.
e) flipflop{52b} : flip symbol{31b} ; flop symbol{31b}.

{Examples:
a) 1 ; true ; false ; f ; : ; 1 ; . ;
b) 1 ; . ; $_{10}$;
c) 1 ;
d) 1 ;
e) 1 ; 0 }

{Denotations-tokens are constituents of denotations (5.0.1.a). Some denotations-tokens may, by themselves, be denotations, e.g. the digit-token 1, whereas others, e.g. the routine-symbol, serve only to construct denotations. }

3.0.4. Action tokens

a)* action token : operator token{b} ; equals symbol{31c} ;
    dereference symbol{31c} ; confrontation token{d}.
b) operator token{42e} : minus and becomes symbol{31c} ;
    plus and becomes symbol{31c} ; times and becomes symbol{31c} ;
    over and becomes symbol{31c} ; modulo and becomes symbol{31c} ;
    plus and becomes symbol{31c} ; or symbol{31c} ; and symbol{31c} ;
    differs from symbol{31c} ; is less than symbol{31c} ;
    is at most symbol{31c} ; is at least symbol{31c} ;
    is greater than symbol{31c} ; plusminus{c} ; times symbol{31c} ;
    quotient symbol{31c} ; modulo symbol{31c} ; over symbol{31c} ;
    element symbol{31c} ; to the power symbol{31c} ;
    lower bound symbol{31c} ; upper bound symbol{31c} ;
    lower state symbol{31c} ; upper state symbol{31c} ;
    plus i times symbol{31c} ; absolute value of symbol{31c};

3.0.4. continued

representation symbol{31c} ; not symbol{31c} ;
lengthen symbol{31c} ; shorten symbol{31c} ; odd symbol{31c} ;
sign symbol{31c} ; round symbol{31c} ; entier symbol{31c} ;
real part of symbol{31c} ; imaginary part of symbol{31c} ;
conjugate symbol{31c} ; binal symbol{31c} ;
booleans to bits symbol{31c} ; characters to bytes symbol{31c} ;
down symbol{31c} ; up symbol{31c}.

c) plusminus{512h, 55p} : plus symbol{31c} ; minus symbol{31c}.
d)* confrontation token : becomes symbol{31c} ;
conforms to symbol{31c} ; conforms to and becomes symbol{31c} ;
is symbol{31c} ; is not symbol{31c}.

{Examples:
a) + ; = ; val ; := ;
b) minus ; plus ; times ; over ; modb ; prus ; ∨ ; ∧ ; ≠ ;
< ; ≤ ; ≥ ; > ; + ; × ; ÷ ; ÷: ; / ; elem ; ↑ ; lwb ; upb ;
lws ; ups ; ⊥ ; abs ; repr ; ¬ ; leng ; short ; odd ; sign ;
round ; entier ; re ; im ; conj ; bin ; btb ; ctb ; down ; up ;
c) + ; − ;
d) := ; :: ; ::= ; :=: ; :≠: }

{Operator-tokens are constituents of formulas (8.4.1.a). An operator-token may be caused to possess an operation by the elaboration of an operation-declaration (7.5.1.a). Confrontation-tokens are constituents of confrontations (8.3.0.1.a). }

3.0.5. Declaration tokens

a)* declaration token : PRIMITIVE symbol{31d} ; long symbol{31d} ;
structure symbol{31d} ; reference to symbol{31d} ;
flexible symbol{31d} ; either symbol{31d} ; procedure symbol{31d} ;
union of symbol{31d} ; mode symbol{31d} ; complex symbol{31d} ;
bits symbol{31d} ; bytes symbol{31d} ; string symbol{31d} ;
file symbol{31d} ; priority symbol{31d} ; local symbol{31d} ;
operation symbol{31d}.

3.0.5. continued

{Examples:

a) int ; long ; struct ; ref ; flex ; either ; proc ; union ; mode ; compl ; bits ; bytes ; string ; file ; priority ; loc ; op }

{Declaration-tokens either are or are constituents of declarers (7.1.1.a), which specify modes (2.2.4), or of declarations (7.2.1.a, 7.3.1.a, 7.4.1.b, 7.5.1.b). }

3.0.6. Syntactic tokens

a)* syntactic token : open symbol{31e} ; close symbol{31e} ;
    comma symbol{31e} ; parallel symbol{31e} ; sub symbol{31e} ;
    bus symbol{31e} ; up to symbol{31e} ; at symbol{31e} ;
    if symbol{31e} ; THELSE symbol{31e} ; fi symbol{31e} ;
    of symbol{31e} ; void symbol{31e} ; label symbol{31e}.

{Examples:

a) (;) ; , ; par ; [;] ; : ; at ; if ; then ; fi ; of ; void ; : }

{Syntactic-tokens separate external objects or group them together.}

3.0.7. Sequencing tokens

a)* sequencing token : go on symbol{31f} ; completion symbol{31f} ;
    go to symbol{31f}.

{Examples:

a) ; ; . ; go to }

{Sequencing-tokens are constituents of clauses, in which they specify the order of elaboration (6.1.1.c,d,j,l, 8.2.7.1.d). }

3.0.8. Hip tokens

a)* hip token : skip symbol{31g} ; nil symbol{31g}.

{Examples:
a) skip ; nil }

{Hip-tokens function as skips and nihils (8.2.7.1.c,e).}

3.0.9. Extra tokens and comments

a)* extra token : for symbol{31h} ; from symbol{31h} ; by symbol{31h} ;
to symbol{31h} ; while symbol{31h} ; do symbol{31h} ;
then if symbol{31h} ; else if symbol{31h} ;
case symbol{31h} ; in symbol{31h} ; esac symbol{31h}.

b) comment{9.1} : comment symbol{31i},
comment item{c}sequence option, comment symbol{31i}.

c) comment item{b} : character token{d} ;
other comment item{1.1.5.C}.

d) character token{c,53c} : LETTER{302b} ; number token{303b} ;
plus i times symbol{31c} ; open symbol{31e} ;
close symbol{31e} ; space symbol{31b} ; comma symbol{31e}.

{Examples:
a) for ; from ; by ; to ; while ; do ; thef ; elsf ; case ; in ;
esac ;
b) c with respect to c ;
c) w ; ? ;
d) a ; 1 ; i ; (;) ; . ; , }

{Extra-tokens and comments may occur in constructions which, by virtue of the extensions of Chapter 9, stand for constructions in which no extra-tokens or comments occur. Thus, a program containing an extra-token or a comment is necessarily a program in the extended language, but not conversely.}

3.0.10. Special tokens

a)* special token : quote symbol{31j} ; comment symbol{31j} ;
indicant{1.1.5.b}.

{Examples:
a) " ; <u>c</u> ; ? }

3.1. Symbols

3.1.1. Representations

a) Letter tokens

symbol	representation	symbol	representation
letter a symbol{302b}	a	letter n symbol{302b}	n
letter b symbol{302b}	b	letter o symbol{302b}	o
letter c symbol{302b}	c	letter p symbol{302b}	p
letter d symbol{302b}	d	letter q symbol{302b}	q
letter e symbol{302b}	e	letter r symbol{302b}	r
letter f symbol{302b}	f	letter s symbol{302b}	s
letter g symbol{302b}	g	letter t symbol{302b}	t
letter h symbol{302b}	h	letter u symbol{302b}	u
letter i symbol{302b}	i	letter v symbol{302b}	v
letter j symbol{302b}	j	letter w symbol{302b}	w
letter k symbol{302b}	k	letter x symbol{302b}	x
letter l symbol{302b}	l	letter y symbol{302b}	y
letter m symbol{302b}	m	letter z symbol{302b}	z

3.1.1. continued

b) Denotation tokens

symbol	representation
digit zero symbol{303d}	0
digit one symbol{303d, 73b}	1
digit two symbol{303d, 73c}	2
digit three symbol{303d, 73d}	3
digit four symbol{303d, 73e}	4
digit five symbol{303d, 73f}	5
digit six symbol{303d, 73g}	6
digit seven symbol{303d, 73h}	7
digit eight symbol{303d, 73i}	8
digit nine symbol{303d, 73j}	9
point symbol{303b, 512d, 553c}	.
times ten to the power symbol{303b, 512g}	$_{10}$ e
true symbol{513a}	true
false symbol{513a}	false
formatter symbol{55a}	f
routine symbol{54b}	: expr
flip symbol{303e}	1
flop symbol{303e}	0
space symbol{309d}	.

c) Action tokens

symbol	representation
minus and becomes symbol{304b}	minus
plus and becomes symbol{304b}	plus
times and becomes symbol{304b}	times
over and becomes symbol{304b}	over
modulo and becomes symbol{304b}	modb
prus and becomes symbol{304b}	prus

3.1.1. continued 2

symbol	representation		
or symbol{304b}	∨	or	
and symbol{304b}	∧	and	
differs from symbol{304b}	≠	ne	
is less than symbol{304b}	<	lt	
is at most symbol{304b}	≤	le	
is at least symbol{304b}	≥	ge	
is greater than symbol{304b}	>	gt	
times symbol{304b}	×	*	
quotient symbol{304b}	÷	quotient	
modulo symbol{304b}	÷:	mod	
over symbol{304b}	/		
element symbol{304b}	elem		
to the power symbol{304b}	↑	power	*:
lower bound symbol{304b}	lwb		
upper bound symbol{304b}	upb		
lower state symbol{304b}	lws		
upper state symbol{304b}	ups		
plus i times symbol{304b}	⊥	i	
absolute value of symbol{304b}	abs		
representation symbol{304b}	repr		
not symbol{304b}	¬	not	
lengthen symbol{304b}	leng		
shorten symbol{304b}	short		
odd symbol{304b}	odd		
sign symbol{304b}	sign		
round symbol{304b}	round		
entier symbol{304b}	entier		
real part of symbol{304b}	re		
imaginary part of symbol{304b}	im		
conjugate symbol{304b}	conj		
binal symbol{304b}	bin		
booleans to bits symbol{304b}	btb		
characters to bytes symbol{304b}	ctb		
down symbol{304b}	down		
up symbol{304b}	up		

3.1.1. continued 3

symbol	representation	
plus symbol{304c}	+	
minus symbol{304c}	-	
equals symbol{42e,72a,73a,74a,75a}	=	eq
dereference symbol{84h}	val	
becomes symbol{831b}	:=	←
conforms to symbol{832b}	::	ct
conforms to and becomes symbol{832b}	::=	ctab
is symbol{833b}	:=:	is
is not symbol{833b}	:≠:	is not isnot

d) Declaration tokens

symbol	representation
integral symbol{71c}	int
real symbol{71c}	real
boolean symbol{71c}	bool
character symbol{71c}	char
format symbol{71c}	format
long symbol{42c,e,f,510b,52a,71d}	long
structure symbol{71e,k}	struct
reference to symbol{71l,m,n}	ref
flexible symbol{71s,u}	flex
either symbol{71u}	either
procedure symbol{71v}	proc
union of symbol{71y}	union
mode symbol{72a}	mode
complex symbol{42c}	compl
bits symbol{42c}	bits
bytes symbol{42c}	bytes
string symbol{42c}	string
file symbol{42c}	file
priority symbol{73a}	priority
local symbol{851b}	loc
operation symbol{75b}	op

3.1.1. continued 4

e) Syntactic tokens

symbol	representation	
open symbol{21a,30b,309d,54b,554b}	(	<u>begin</u>
close symbol{21a,30h,309d,54b,554b}	)	<u>end</u>
comma symbol{30e,309d,54f,554b,62e,g,71f,p,z,861b,c}	,	<u>comma</u>
parallel symbol{62b,c,d,f}	<u>par</u>	
sub symbol{71o,861a}	[	(
bus symbol{71o,861a}	]	)
up to symbol{71q,861f}	:	
at symbol{861g}	<u>at</u>	
if symbol{64a}	(	<u>if</u>
then symbol{64e}	|	<u>then</u>
else symbol{64e}	|	<u>else</u>
fi symbol{64a}	)	<u>fi</u>
of symbol{852a}	<u>of</u>	
void symbol{823b}	<u>void</u>	
label symbol{61k}	:	

f) Sequencing tokens

symbol	representation	
go on symbol{30c,54f,61c,d,j}	;	
completion symbol{61l}	.	<u>exit</u>
go to symbol{827d}	<u>go to</u>	<u>goto</u>

g) Hip tokens

symbol	representation
skip symbol{827c}	<u>skip</u>
nil symbol{827e}	<u>nil</u>

h) Extra tokens

symbol	representation	
for symbol {9.3.a,b}	for	
from symbol {9.3.a,b,c}	from	
by symbol {9.3.a,b,c}	by	
to symbol {9.3.a,c}	to	
while symbol {9.3.a,b,c}	while	
do symbol {9.3.a,b,c}	do	
then if symbol {9.4.a}	|:	thef
else if symbol {9.4.a,b}	|:	elsf
case symbol {9.4,b,c,d,e}	(	case
in symbol {9.4.b,c,d,e}	|	in
esac symbol {9.4.b,c,d,e}	)	esac

i) Special tokens

symbol	representation	
quote symbol{514a,53a,b,d}	"	
comment symbol{309b}	c	comment

3.1.2. Remarks

a) Where more than one representation of a symbol is given, any one of them may be chosen. {However, discretion should be exercised, since the text (a > b then b | a fi, though acceptable to an automaton, would be more intelligible to a human in either of the two representations

(a> b | b | a)

or

if a > b then b else a fi.}

3.1.2. continued

b) A representation which is a sequence of underlined or bold-faced marks or a sequence of marks between apostrophes is different from the sequence of those marks when not underlined, in bold face or between apostrophes.

c) Representations of other terminal productions of 'letter token' {1.1.4.Step 2}, 'indicant' {1.1.5.b}, "other comment item' and 'other string item' {1.1.5.c} may be added, provided that no sequence of representations of symbols can be confused with any other such sequence . {e.g., <u>do</u> <u>if</u> are representations of the do-symbol followed by the if-symbol, whereas <u>doif</u> might be an ill-chosen representation of an other-indication.}

d) The fact that representations of the terminal productions of 'letter token' given above are usually spoken of as small letters is not meant to imply that the so-called corresponding capital letters could not serve equally well as representations. On the other hand, if both a small letter and the corresponding capital letter occur, then one of them is the representation of an other terminal production of 'letter-token' {1.1.4.Step 2}.

{For certain different symbols, one same representation is given, e.g. for the routine-symbol, up-to-symbol and label-symbol, the representation ":" is given. It follows uniquely from the syntax which of these three symbols is represented by an occurrence of ":" outside comments and row-of-character-denotations. Also, some of the given representations appear to be "composite"; e.g., the representation ":=" of the becomes-symbol appears to consist of ":", which looks like the representation ":" of the routine-symbol, etc., and the representation "=" of the equals-symbol. It follows from the Syntax that ":=" or even ":=" can occur outside comments and row-of-character-denotations as representation of the becomes-symbol only (since "=" cannot occur as representation of a monadic-operator). Similarly, the other given composite representations do not cause ambiguity. }

4. Identification and the context conditions

{A proper program is a program satisfying the context conditions, e.g., if (real x ; x := 1) is contained in a proper-program, then the second occurrence of x is a reference-to-real-mode-identifier not solely because of some production rule (though this might be possible with a more elaborate syntax) but also because it identifies the first occurrence according to one of the context conditions. This chapter describes the methods of identification and contains other context conditions which prevent such undesirable constructions as mode a = a.}

4.1. Identifiers

{Identifiers are sequences of letter-tokens and/or digit-tokens in which the first is a letter-token, e.g. x1. ~~Identifiers, except for label~~ Mode-identifiers, are made to possess values by the elaboration of identity-declarations (7.4.1.a). Some mode-identifiers possessing values which are not names might, in other languages, be called constants, e.g. m in int m = 4096. Identifiers possessing names which refer to such values might be called variables and those possessing names which refer to names might be called pointers. Such terminology is not used in this Report. Here, all ~~identifiers, except for label~~mode-identifiers, possess values, which are or are not names.}

4.1.1. Syntax

a)* identifier : MABEL identifier{b}.
b) MABEL identifier{54g,61k,827d,860a} : TAG{c,d,302b}.
c) TAG LETTER{b,c,d,71j} : TAG{c,d,302b}, LETTER{302b}.
d) TAG DIGIT{b,c,d,71j} : TAG{c,d,302b}, DIGIT{303d}.
e)* range : SORTETY serial CLAUSE{61a} ; PROCEDURE denotation{54b}.

{Examples:
b) x ; xx ; x1 ; amsterdam }

{Rule b together with 1.2.2.v and 1.2.1.l gives rise to an infinity of production rules of the strict language, one for each pair of terminal productions of 'MABEL' and 'TAG'.

For example,

'real$\overset{\text{mode}}{\wedge}$identifier : letter a letter b.'

is one such production rule. From rule c and 3.0.2.b, one obtains

'letter a letter b : letter a, letter b.',

'letter a : letter a symbol.' and

'letter b : letter b symbol.',

yielding

'letter a symbol, letter b symbol'

as a terminal production of a 'real$\overset{\text{mode}}{\wedge}$identifier'. For additional insight into the function of rules c and d, see 7.1.1.h and 8.5.2}

4.1.2. Identification of identifiers

{The method of identification is first to distinguish between defining and applied occurrences of terminal productions of 'MABEL identifier' and then to discover which defining occurrence is identified by a given applied occurrence. }

a) A given occurrence of a terminal production of 'MABEL identifier' where "MABEL" stands for any terminal production of the metanotion 'MABEL' is a defining occurrence if it follows a formal-declarer {7.1.1.b}, or if it is contained in a label {6.1.1.k} ; otherwise, it is an "applied occurrence".

b) If a given occurrence of a terminal production of 'MABEL identifier' (see a) is an applied occurrence, then it may identify a defining occurrence of the same terminal production found by the following steps:

Step 1: The given occurrence is termed the "home" and Step 2 is taken;

Step 2: If there exists a smallest range containing the home, then this range, with the exclusion of all ranges contained within it, is termed the home and Step 3 is taken {; otherwise, there is no defining occurrence which the given occurrence identifies; see 4.4.1.b};

Step 3: If the home contains a defining occurrence of the same terminal production of 'MABEL identifier', then the given occurrence identifies it; otherwise, Step 2 is taken.

{In the closed-clause (string s := "abc" ; s[3] ≠ "d"), the first occurrence of s is a defining occurrence of a terminal production of 'reference-to-row-of-character-mode-identifier'. The second occurrence of s identifies the first and, in order to satisfy the identification condition (4.1.1.a), is also a terminal production of a'reference-to-row-of-boolean-identifier'. Identifiers have no inherent meaning. }

4.2. Indications

{Indications are used for modes, priorities and operators. The representation of indications chosen in this Report are sequences of bold-faced or underlined letters, e.g. compl and plus, but no production rule determines this sequence. The programmer may also create his own indications, with suitable representations, provided that they cannot be confused with an other symbol (1.1.5.b, 3.1.2.c). }

4.2.1. Syntax

a)* indication : MODE mode indication{b} ; ADIC indication{e,f}.
b) MODE mode indication{71b} : mode standard{c} ; ~~other~~ indicant{1.1.5.b}.
c) mode standard{b} : string symbol{31d} ; file symbol{31d} ;
 long symbol{31d} sequence option, complex symbol{31d} ;
 long symbol{31d} sequence option, bits symbol{31d} ;
 long symbol{31d} sequence option, bytes symbol{31d}.
d)* dyadic indication : PRIORITY indication{e}.
e) PRIORITY indication{43b,73a} :
 long symbol{31d} sequence option, operator token{304b} ;
 long symbol{31d} sequence option, equals symbol{31c} ;
 ~~other~~ indicant{1.1.5.b}.
f) monadic indication{43c} :
 long symbol{31d} sequence option, operator token{304b} ;
 ~~other~~ indicant{1.1.5.b}.
g)* adic indication : ADIC indication{e,f}.

{Examples:
b) compl ; primitive ;
c) string ; file ; long compl ; bits ; long bytes ;
e) + ; = ; ? ;
f) + ; long btb ; ? }

4.2.2. Identification of indications

{The identification of indications is similar to that of identifiers.}

a) A given occurrence of a terminal production of 'MODE mode indication' ('PRIORITY indication') where "MODE"("PRIORITY") stands for any terminal production of the metanotion 'MODE' ('PRIORITY') is an indication-defining occurrence if it precedes the equals-symbol of a mode-declaration {7.2.1.a} (priority-declaration{ 7.3.1.a}); otherwise, it is an "indication-applied-occurrence".

b) If a given occurrence of a terminal production of 'MODE mode indication' ('PRIORITY indication') (see a) is an indication-applied occurrence, then it may identify an indication-defining occurrence of the same terminal production found by using the steps of 4.1.2.b with Step 3 replaced by: "Step 3: If the home contains an indication-defining occurrence of the same terminal production of 'MODE mode indication' ('PRIORITY' indication'), then the given occurrence identifies it;
otherwise, Step 2 is taken."

{Indications have no inherent meaning.
A terminal production of 'monadic indication' has no indication-defining occurrence. }

4.3. Operators

{Operators are either monadic, i.e. require a right operand only, or are dyadic, i.e. require both a left and a right operand, e.g. abs x and x + y. Operators are made to possess routines by the elaboration of operation-declarations (7.5.1.a).
Operators are identified by observing the modes of their operands, e.g. x ÷ y, x + i, i + x, i + j each involves a different operator, see 10.2.3.i, 10.2.4.a, 10.2.4.b and 10.2.2.i. Though an operator knows the mode of the value, if any, delivered by its routine, this mode is not involved in the identification process. }

4.3.1.Syntax

a)* operator : PRAM ADIC operator{b,c}.

b) procedure with LMODE parameter and RMODE parameter MOID PRIORITY
   operator{75b,84b} : PRIORITY indication{42e}.

c) procedure with RMODE parameter MOID monadic operator{75b,84g} :
   monadic indication{42f}.

d)* dyadic operator : procedure with LMODE parameter and RMODE
   parameter MOID PRIORITY operator{b}.

e)* monadic operator :
   procedure with RMODE parameter MOID monadic operator{c}.

{Examples:

b) +

c) abs }

4.3.2. Identification of operators

{The identification of operators is similar to that of identifiers and indications, except that different occurrences of one same terminal production of 'ADIC indication' may be occurrences of more than one terminal production of 'PRAM ADIC operator' and, therefore, the modes of the operands must be considered. }

a) A given occurrence of a terminal production of 'PRAM ADIC operator' where "PRAM" ("ADIC") stands for any terminal production of the metanotion 'PRAM' ('ADIC') is an operator-defining occurrence if it precedes the equals-symbol of an operation-declaration {7.5.1.a} ; otherwise, it is an "operator-applied occurrence".

b) If a given occurrence of a terminal production of 'PRAM ADIC operator' (see a) is an operator-applied occurrence, in a formula {8.4.1.a}, than it may identify an operator-defining occurrence of the same terminal production found by using the steps of 4.1.2.b, with Step 3 replaced by :
"Step 3: If the home contains an operator-defining occurrence, in an operation-declaration {7.5.1.a,b} of a terminal production of 'PRAM ADIC operator' which is the same terminal production of 'ADIC indication' as the given occurrence, and which is such that the terminal production of the

4.3.2. continued

metanotion 'LMODE' ('RMODE') (see 8.4.1.b,g) of the left (right) operand of that formula can be firmly coerced to {4.4.3.a} the mode specified by the first (second) virtual-parameter{7.1.1.x} contained in the virtual-plan{7.5.1.b} of that operation-declaration, then the given occurrence identifies that operator-defining occurrence; otherwise, Step 2 is taken."

{Operators have no inherent meaning; an operator-defining occurrence is made to possess a routine (2.2.3.4) by the elaboration of an operation-declaration (7.5.1.a).

A given indication may be both a ~~priority~~ dyadic-indication and a ~~priority~~ dyadic-operator. As a ~~priority~~ dyadic-indication, it identifies its indication-defining occurrence. As a ~~priority~~ dyadic-operator, it may identify an operator-defining occurrence, which possesses a routine. Since the indication preceding the equals-symbol of an operation-declaration is an indication-application and an operator-definition (but not an operator-application), it follows that the set of those occurrences which identify a given ~~priority~~ dyadic-operator is a subset of those occurrences which identify the same ~~priority~~ dyadic-indication.

In the closed-clause

```
begin real x,y := 1.5 ; priority min = 6 ;
op min = (real a, b)real : (a > b | b | a) ;
x := y min pi / 2 end ,
```

the first occurrence of min is an indication-defining priority-SIX-indication. The second occurrence of min is indication-applied and identifies the first occurrence (4.2.2), whereas, at the same textual position, min is also operator-defined as a [prrr]-priority-SIX-operator ~~and hence is also a [illegible]-priority-SIX-operator (7.5.1.c) [illegible], ignoring the mode of the value [illegible] which is delivered)~~, where "[prr]" stands for 'procedure-with-real-parameter-and-real-parameter', and "[prrr]" for '[prr]-real'. The third occurrence of min is indication-applied and, as such, identifies the first occurrence, whereas, at the same textual position, min is also operator-applied, and, as such, identifies the second occurrence; this makes it ~~(in view of Step 3) a [illegible]-priority-SIX-operator and hence,~~ because of the identification condition (4.4.1.a), a [prrr]-priority-SIX-operator. This identification of the priority-operator is made because:

i) min occurs in an operation-declaration,

ii) the base y can be firmly coerced to the mode specified by real,

iii) the formula pi / 2 is a priori of the mode specified by <u>real</u>,

iv) min is thus ~~a [prrr]-priority-SIX-operator, and~~

~~v)~~ because of the identification condition ~~as is thus also~~ a [prrr]-priority-SIX-operator.

If the ~~identification~~ first three conditions were not satisfied, then the search for another defining occurrence would be continued in the same range, or failing that, in a surrounding range. }

{Though this be madness, yet
there is method in't.
Hamlet, William Shakespeare.}

4.4. Context conditions

A "proper" program is a program satisfying the context conditions; a "meaningful" program is a proper program whose elaboration is defined by this Report {Whether all programs, only proper programs, or only meaningful programs are "ALGOL 68" programs is a matter for individual taste. If one chooses only proper programs, then one may consider the context conditions as syntax which is not written as production rules. }

4.4.1. The identification conditions

a) In a proper program, a defining (indication-defining, operator-defining) occurrence of a terminal production of a notion ending on 'identifier' ('indication', 'operator') and each applied (indication-applied, operator-applied) occurrences identifying it are occurrences of one same notion ending on 'identifier' ('indication', 'operator').

---

b) No proper program contains an applied (indication-applied, operator-applied) occurrence of a terminal production of a notion ending on 'identifier' ('indication', 'operator') which does not identify a defining (indication-defining, operator-defining) occurrence.

4.4.2. The uniqueness condition

a) A "reach" is a range {4.1.1.e} with the exclusion of all its constituent ranges.

b) No proper program contains a reach{a} containing two defining (indication-defining) occurrences of a given terminal production of a notion ending on 'identifier' ('indication').

{e.g., none of the claused-clauses (6.4.1.a)

```
(real x, real x ; sin(3.14)),
(real y ; int y ; sin(3.14)),
(real p ; p: go to p ; sin(3.14)),
(mode a = real ; mode a = bool ; sin(3.14)),
(priority b = 5 ; priority b = 6 ; sin(3.14))
```

is contained in a proper program. }

c) No proper program contains a reach containing two operation-declarations whose first constituent operators are the same terminal productions of a notion ending on 'indication' and all of whose corresponding constituent virtual-parameters {7.5.1.b, 5.4.1.c,d,e, 7.1.1.x} are virtual-declarers specifying modes loosely related to one another {4.4.3.c}.

{e.g., neither the closed-clause

```
(op max = (int a, int b)int : (a> b | a | b) ;
 op max = (int a, int b)real : (a> b | a | b) ; sin(3.14))
```

nor

```
(op max = (int a, ref int b)int : (a > b | a | b) ;
 op max = (ref int a, int b)int : (a > b | a | b) ; sin(3.14))
```

is contained in any proper program, but

```
(op max = (int a, int b)real : (a > b | a | b) ;
 op max = (real a, real b)real : (a > b | a | b) ; sin(3.14))
```

may be. }

{In the pragmatic remarks in the sequel, "in the reach of the declaration" stands for "in a position where all identifications are as in a reach containing the declaration". }

4.4.3. The mode conditions

a) A given mode is "firmly coerced from" ("united from") a second mode if the notion consisting of that second mode followed by 'base' is a production of the notion consisting of 'firm' ('strongly united to') followed by the given mode followed by 'base' {see 8.2}.

{e.g., the mode specified by real is firmly coerced from the mode specified by ref real because the notion 'reference to real base' is a production of 'firm real base' (8.2.0.1.e, 8.2.1.1.a); similarly, that specified by union(int, real) is united from those specified by int and real. }

b) Two modes are "related" to one another if they are both firmly coerced {a} from one same mode. {A mode is related to itself.}

c) Two modes are "loosely related" if they either are related or are 'row of LMODE' and 'row of RMODE' where "LMODE" and "RMODE" stand for different loosely related modes.

{e.g., the modes specified by proc real and ref real are related and, hence, loosely related and those specified by []real and by [] ref real are loosely related but not related.}

d) No proper program contains a declarer {7.1.1.a} specifying a mode united from {a} two modes related {b} to one another.

{e.g., the declarer union(real, ref real) is not contained in any proper program.}

e) No proper program contains a declarer {7.1.1.a} the constituent field-selectors {7.1.1.i} of two of whose constituent field-declarators {7.1.1.g} are the same sequence of symbols.

{e.g., the declarer struct(int i, bool i) is not contained in any proper program, but struct(int i, struct(int i, bool j) j) may be.}

4.4.4. The declaration condition

a) A mode-indication {4.2.1.b} contained in an actual-declarer {7.1.1.b} is "shielded" if

i) it is or is contained in a virtual-declarer {7.1.1.b} following a reference-to-symbol {3.1.1.d} in a field-declarator {7.1.1.g}, or

ii) it is or is contained in a virtual-declarer contained in a field-declarator contained in a virtual-declarer following a reference-to-symbol, or

iii) it is contained in a virtual-parameter {7.1.1.x}, or
iv) it is contained in a virtual-declarer following a virtual-parameters-pack {5.4.1.j}, or
v) it is contained in a strict-lower-bound or strict-upper-bound {7.1.1.u}.

{e.g., person is shielded in struct(int age, ref person father), but not in struct(int age, person uncle) and p is shielded in proc(p)p but not in union(int, []p).}

b) An actual-declarer {7.1.1.b} may "show" a given mode-indication {4.2.1.b} ; this is determined in the following steps:
Step 1: If it is, or contains and does not shield, a mode-indication which is the same terminal production as the given mode-indication, then it shows the given mode-indication; otherwise, Step 2 is taken:
Step 2: If it is, or contains and does not shield an occurrence of a not "encountered" terminal production of 'mode indication', then that terminal production is remembered to have been encountered and the occurrence is replaced by a copy of the actual-declarer of that mode-declaration {7.2.1.a} which contains the indication-defining occurrence {4.2.2.b} identified by the occurrence to be replaced, and Step 1 is taken; otherwise, it does not show the given mode-indication.

{e.g., in the declarations mode a = [1:2]b, b = union(ref d), d = struct(ref e @), e = proc(int)a, the mode-indications shown by [1:2]b are b and d.}

c) No proper program contains a mode-declaration {7.2.1.a} whose mode-indication is shown by its actual-declarer.

{e.g., no proper program contains one of the following declarations:
mode a = a ; mode b = e, e = [1:10]b ; mode d = []ref union(proc(d)d, proc d) ;
mode parson = struct(int age, parson uncle) }

⊥ if necessary, the resulting sequence of symbols is turned into an actual-declarer by replacing the actual-lower-bounds (actual-upper-bounds) {7.1.1.t} in that copy by virtual-lower-bounds (virtual-upper-bounds) {7.1.1.s},

5. Denotations

{Denotations, e.g. 3.14 or "abc" are terminal productions of notions whose value is independent of the elaboration of the program. In other languages they are sometimes called "literals" or "constants".}

5.0.1. Syntax

a)* denotation : PLAIN denotation{510b,51a,511a,512a,513a,514a} ;
 structured with row of boolean field LENGTHETY letter aleph denotation{52a} ; row of character denotation{53a} ;
 PROCEDURE denotation{54b} ; format denotation{55a}.

{Examples:
a) 3.14 ; 101 ; "algol.report" ; ((bool a)int : (a | 1 | 0)) ;
 f5df }

5.0.2. Semantics

a) Each occurrence of a terminal production of a given notion generating a denotation possesses an instance of one same value whose mode is that enveloped by that notion; its elaboration involves no action.

{E.g., The value of "algol.report" which is a production of'row of character denotation', is of the mode 'row of character'.}

5.1. Plain denotations

{Plain-denotations are those of arithmetic, boolean and character values, e.g. 1, 3.14, true and "a".}

5.1.0.1. Syntax

a)* plain denotation : PLAIN denotation{510b,511a,512a,513a,514a}.

b) long INTREAL denotation{860a} :
 long symbol{31d}, INTREAL denotation{511a,512a}.

5.1.0.2. Semantics

a) A plain-denotation possesses a plain value {2.2.3.1} , but plain values possessed by different plain-denotations are not necessarily different {e.g., 123.4 and 1.234e+2}.

b) The value of a denotation consisting of a number {, possibly zero,} of long-symbols followed by an integral-denotation (real-denotation) is the "a priori" value of that integral-denotation (real-denotation) provided that it does not exceed the largest integer {10.1.b} (largest real number{10.1.d}) of length number one more than that number of long-symbols {; otherwise, the value is undefined}.

5.1.1. Integral denotations

5.1.1.1. Syntax

a) integral denotation{860a,510b,512c,d,h,55g} : digit token{303c} sequence.

{Examples:

a) 0 ; 4096 ; 00123 (Note that -1 is not an integral-denotation.)}

5.1.1.2. Semantics

The a priori value of an integral-denotation is the integer which in decimal notation is that integral-denotation in the representation language {1.1.8}. {See also 5.1.0.2.b.}

5.1.2. Real denotations

5.1.2.1. Syntax

a) real denotation{860a,510b} :
variable point numeral{b} ; floating point numeral{e}.

b) variable point numeral{a} :
integral part{c} option, fractional part{d}.

c) integral part{b} : integral denotation{511a}.

5.1.2.1. continued

d) fractional part{b} : point symbol{31b}, integral denotation{511a}.
e) floating point numeral{a} : stagnant part{f}, exponent part{g}.
f) stagnant part{e} : integral denotation{511a} ; variable point numeral{b}.
g) exponent part{e} : times ten to the power symbol{31b}, power of ten{h}.
h) power of ten{g} : plusminus{304c} option, integral denotation{511a}.

{Examples:

a)	0.000123 ; 1.23e-4	b)	.123 ; 0.123 ;
c)	123 ;	d)	.123 ;
e)	1.23e-4	f)	1 ; 1.23 ;
g)	e-4 ;	h)	3 ; +45 ; -678 }

5.1.2.2. Semantics

a) The a priori value of a fractional-part is the a priori value of its integral-denotation devided by ten as many times as there are digit-tokens in the fractional-part.

b) The a priori value of a variable-point-numeral is the sum in the sense of numerical analysis of zero, the a priori value of its integral-part, if any, and that of its fractional-part, if any {see also 5.1.0.2.b}.

c) The a priori value of an exponent-part is ten raised to the a priori value of the integral-denotation in its power-of-ten if that power-of-ten does not begin with a minus-symbol; otherwise, it is one-tenth raised to the a priori value of that integral-denotation.

d) The a priori value of a floating-point-numeral is the product in the sense of numerical analysis of the a priori values of its stagnant-part and exponent-part {see also 5.1.0.2.b}.

5.1.3. Boolean denotations

5.1.3.1. Syntax

a) boolean denotation{860a} : true symbol{31b} ; false symbol{31b}.

5.1.3.1. continued

{Examples:
a) true ; false }

5.1.3.2. Semantics

The value of a true-symbol (false-symbol) is true (false).

5.1.4. Character denotations

5.1.4.1. Syntax

a) character denotation{860a} :
   quote symbol{31i}, string symbol{531b}, quote symbol{31i}.

{Examples:
a) "a" }

5.1.4.2. Semantics

The value of a character-denotation is a new instance of the character possessed {5.3.2.a} by its string-item {5.3.1.b} if that string-item is a character-token or an other-string-item; otherwise, {if that string-item is a quote-image,} then it is a new instance of the character possessed by the quote symbol.

5.2. Bits denotations

{There are two kinds of denotations of structured or multiple values viz., bits, e.g. 1011, and string, e.g. "abc". These denotations differ in that a string denotation contains zero or two or more string-items but a bits denotation may contain one or more flipflops. (See also character-denotations 5.1.4.)}

5.2.1. Syntax

a) structured with row of boolean field LENGTH LENGTHETY letter aleph denotation{860a} : long symbol{31d}, structured with row of boolean field LENGTHETY letter aleph denotation {a,b}.

5.2.1. continued

b) structured with row of boolean field letter aleph denotation{860a} : flipflop{303e} sequence.

{Examples:
a) long 1011 ;
b) 1011 }

5.2.2. Semantics

a) Let "m" stand for the number of flipflops in the denotation and "n" for the value of L bits width {10.1.g}, L standing for as many times long as there are long-symbols in the denotation; if m ≤ n, then the value of the structured-with-row-of-boolean-field-letter-aleph-denotation is a structured value with one field selected by letter-aleph, that field being a multiple value {2.2.3.3} whose descriptor has an offset 1 and one quintuple (1,n,1,1,1) and whose element with index "j" is a new instance of false for j = 1, ... , n-m, and for j = n-m+1, ... , n is a new instance of true (false) if the i-th constituent flipflop (i = j + m - n) of the denotation is a flip-symbol (flop-symbol).

5.3. Row of character denotations

{The denotations of strings always begin and end with a quote-symbol, e.g. "abc". If it is necessary to include a quote within a string, then the quote-symbol is doubled, e.g. "this.is.a.quote.""". Since the syntax nowhere allows string- or character-denotations to follow one another, ambiguities do not arise.}

5.3.1. Syntax

a) row of character denotation{860a} : empty string{b} ;
   quote symbol{31i}, string item{c}, sequence proper, quote symbol{31i}.
b) empty string{a} : quote symbol{31i}, quote symbol{31i}.
c) string item{a} : character token{309d } ;
   quote image{d} ; other string item{1.1.5.c}.
d) quote image{c} : quote symbol{31i}, quote symbol{31i}.

5.3.1. continued

{Examples:

a) "" ; "abc" ; """a.+.b"".is .a. formula ; b) "" ;

c) a ; "" ; ? ; d) "" }

5.3.2. Semantics

a) Each character-token and other-string-item, as well as the quote-symbol {not quote-image} possesses a unique character.

b) The value of a row-of-character-denotation is a multiple value 2.2.3.3 whose descriptor has an offset 1 and one quintuple (1,n,1,1,1), where n stands for the number of string-items contained in the denotation. For i = 1, ... ,n, the element with index i of that multiple value is a new instance of the character possessed by the i-th string-item, and, otherwise, {if that string-item is a quote-image} is a new instance of the character possessed by the quote-symbol.

{The construction "a" is a character-denotation, not a string denotation. However, in all strong positions, e.g. string s := "a", it will be rowed to a multiple value {8.2.6}. Elsewhere, where a multiple value is required, a generator may be used, e.g. as in union(int, string) ns := string := "a".}

5.4. Routine denotations

{A routine-denotation, e.g. ((real a, b)real : (a > b | b | a)), always has a routine-symbol (:). To the left of this symbol stand the formal-parameters, e.g. (real a, b), and a declarer specifying the mode of the value delivered, if any, e.g. real. To the right of the routine-symbol is the body, e.g. (a > b | b | a), which is a unitary-clause. The whole is enclosed between an open-symbol and a close-symbol, but they may often be omitted, see the extension 9.2.d. It is essential that, in general, a routine-denotation be closed, for otherwise denotations like (int sintzoff)void : (int branquart)void : lewi (wodon) could also be calls, or formulas like (int a)int : 1 + 2 + 3 would be ambiguous if + is also declared as an operator accepting a routine as left operand. }

5.4.1. Syntax

a)* routine denotation : PROCEDURE denotation{b}.
b) procedure PARAMETY MOID denotation{860a} :
    open symbol{31e}, formal procedure PARAMETY MOID plan{c,d},
    routine symbol{31b},MOID body{i}, close symbol{31e}.
c) VICTAL procedure with PARAMETERS MOID plan{b,75b,71w} :
    VICTAL PARAMETERS{e,g,71x,74b} pack, virtual MOID declarer{g,71b}.
d) VICTAL procedure MOID plan{b,71w} : virtual MOID declarer{h,71b}.
e) VICTAL PARAMETERS and PARAMETER{c,b,862a} :
    VICTAL PARAMETERS{e,g,71x,74b}, semicomma{f},
    VICTAL PARAMETER{g,71x,74b}.
f) semicomma{e} : comma symbol{31e} ; go on symbol{31f}.
g) formal MODE parameter{c,e,74a} :
    formal MODE declarer{71b}, MODE identifier{41b}.
h) virtual void declarer{c,d} : EMPTY.
i) MOID body{b} : strong MOID unit{61e}.
j)* VICTAL parameters pack : VICTAL PARAMETERS{e,f,71y,74b} pack.

{Examples:
b) ((<u>bool</u> a, b) <u>bool</u> : (a | b | <u>false</u>)) ; (: x := 3.14) ;
c) (<u>bool</u> a, b)<u>bool</u> ;
d) <u>void</u> ;
e) <u>bool</u> a, <u>bool</u> b ;
f) , ; ; ;
g) <u>bool</u> a ;
i) (a | b | <u>false</u>) ; x := 3.14}

5.4.2. Semantics

A routine-denotation possesses that routine which can be obtained from it in the following steps:

Step 1: A copy is made of the routine-denotation ;

Step 2: If the routine denotation does not contain a formal-parameters-pack, then Step 3 is taken; otherwise, an equals-symbol followed by a skip-symbol is inserted in the copy following the last identifier in each copied constituent formal-parameter of that formal-parameters-pack; the open-symbol of that formal-parameters-pack is deleted and its close-symbol is replaced by a go-on-symbol ;

Step 3: If the virtual-declarer of its formal-plan is empty, then the routine-symbol which follows it is deleted; otherwise, the routine-symbol is replaced by a becomes-symbol, and an open-symbol followed by a dereference-symbol is placed before and a close-symbol is placed after the copy ;

Step 4: An open-symbol is placed before and a close-symbol is placed after the copy, and the copy, thus modified, is the routine possessed by the routine-denotation.

{The routine possessed by p1 after the elaboration of proc p1 = : tirrenia (, in the strict language, proc p1 = (: tirrenia)) is (tirrenia); that possessed by p2 after the elaboration of proc p2 = real : xx is (val(real := xx)); that possessed by p3 after the elaboration of proc p3 = (int a)real: (a > 0 | xx | yy), is (val(int a = skip ; real := (a > 0 | xx | yy))), and that possessed by p4 after the elaboration of proc p4 = (real a,b) : (a > b | stop) is (real a = skip, real b = skip ; (a > b | stop)). A routine is the same sequence of symbols as some closed-clause (6.3.1). For the use of routines, see 8.4 (formulas), 8.2.2 (deprocedured-coercends) and 8.6.2(calls). }

5.5. Format denotations

5.5.1. Syntax

a) format denotation{860a} : formatter symbol{31b}, collection{b} list, formatter symbol{31b}.

b) collection{a,b} : picture{c} ; insertion{d} option, replicator{f} , collection{b} list pack, insertion{d} option.

c) picture{b} : MODE pattern{552a,553a,554a,555a,556a,557a,-} option, insertion{d} option.

5.5.1. continued

d) insertion{b,c,m,552b,f,554a,557a} : literal{j} option,
   insert{e} sequence ; literal{j} .

e) insert{d} : replicator{f},alignment{i} , literal{j} option.

f) replicator{b,e,j,n} : replication{f} option.

g) replication{f,k,557a} : dynamic replication{h} ;
   integral denotation{51a}.

h) dynamic replication{g} : letter n{302b}, strong integral
   unit{612e}pack.

i) alignment{e} :letter k{302b} ; letter x{302b} ; letter y{302b} ;
   letter l{302b} ; letter p{302b}.

j) literal{d,e,552f,554b} : replicator{f}, STRING denotation{514a,531a},
   replicated literal{k} sequence option.

k) replicated literal{j} : replication{g}, STRING denotation{514a,531a}.

{Examples:

a) fp"table.of"x10a,n(lim-1)(16x3zd,3x10(2x+.12de+2d"+j×"si+.10de+2d))pf ;

b) p"table.of"x10a ; 3x10(2x+.12de+2d"+j×"si+.10de+2d) ;

c) 120kc("mon","tues","wednes","thurs","fri","satur","sun")"day" ; p ;

d) p"table.of"x ; "day" ;

e) p"table.of" ;

g) n(lim-1) ; 10 ;

h) n(lim-1) ;

j) "+j×" ;

k) 20"." }

l) sign mould{552a,553a,d,e} : loose replicatable zero frame{m},
   sign frame{p} ; loose sign frame{m}.

m) loose ANY frame{l,552d,553b,d,555a,556a,557a} :
   insertion{d} option, ANY frame{n,p,q,557b}.

n) replicatable ANY frame{m} : replicator{f}, ANY frame{o,q}.

o) zero frame{n,552e} : letter z{302b}.

p) sign frame{l,m} : plusminus{304c}.

q) suppressible ANY frame{n,m,557a} : letter s{302b}option,
   ANY frame{552e,553c,f,555b,556b}.

r) * frame : ANY frame.

{Examples:

l) "="12z+ ; 2x+ ;

m) "="12z ;

n) 12z ;

q) si ; 10a }

{aa) Three ways of "transput" (i.e. "input" and "output") are provided by the standard-prelude, viz. formatless transput (10.5.2), formatted transput (10.5.3) and binary transput (10.5.4). Formats are used by the formatted transput routines to control input from and output to a "file" (10.5.1). No section on semantics of format-denotations is given, since this is entirely dealt with by the standard-prelude.

bb) A format may be associated with a file by a call of format (10.5.3.a), outf (10.5.3.1.a) or inf (10.5.3.2.a), thereby causing the transformat which is that format to be elaborated (5.5.8.1.b), the collection-list which is the same sequence of symbols as the resulting "string" (i.e. value of mode 'row-of-character') to be unfolded (cc), the resulting picture-list to be the current format of the file and its first constituent picture to be the current picture of the file (; e.g., after the call format (f1,fpt,3(3d.d)1f) the current format of the file f1 is pt, 3d.d, 3d.d, 3d.dl and the current picture is pt).

cc) The result of unfolding a collection-list (10.5.3.b) is a picture list:

a) if the collection-list is a picture, then the result consists of that picture;

b) if the collection-list is a collection but not a picture, then the result consists of the first insertion-option of the collection, followed by as many copies of the result of unfolding the collection-list of its collection-list pack as is the value of its replicator, separated by comma-symbols, followed by its last insertion-option (; e.g., the result of unfolding 3k"ab"2(10a)l is 3k"ab"10a,10al) ;

c) if the collection-list is a collection-list-proper, then the result consists of the result of unfolding the collection of that collection-list-proper, followed by a comma-symbol, followed by the result of unfolding its collection-list (; e.g., the result of unfolding 10a,pn(i)(d.2d)"." is 10a,p"." when the value of i is zero).

dd) When one of the formatted transput routines outf (10.5.3.1.a), out (10.5.3.1.b), inf (10.5.3.2.a) or in (10.5.3.2.b) is called, then transput takes place in the following steps:
Step 1: The values to be transput are elaborated collaterally and the result is straightened (10.5.0) into a series of values, the first of which, if any, is made to be the current value;
Step 2: If the current picture of the file is an insertion, then that insertion is performed (gg), the next picture, if any, is made to be the current picture of the file and Step 2 is taken; otherwise, Step 3 is taken;
Step 3: If the series of values is empty or exhausted, then the transput is accomplished; otherwise, if the picture-list is exhausted, then formatend of the file is called, a routine which may be provided by the programmer (10.5.1.kk);
Step 4: If the current value is compatible with (nn) the current picture, then that value is transput under control of that picture; otherwise, value error of the file is called, a routine which may be provided by the programmer;
Step 5: The next value, if any, is made to be the current value, the next picture, if any, is made to be the current picture and Step 2 is taken.

ee) The value of the empty replicator is one; the value of a replication which is an integral-denotation is the value of that denotation; the value of a dynamic-replication is the value of its strong-integral-unit if that value is positive, and zero otherwise.

ff) Transput occurs at the current "position" (i.e. page number, line number and char number) of the file. At each position of the file within certain limits (10.5.1.1.j,k,l) some character is "present", depending on the contents of the file and on its "conversion-key" (10.5.1.11).

gg) An insertion is performed by performing its constituent alignments and, on output (input), "writing" ("requiring") its constituent literals one after the other.

hh) Performing an alignment affects the position of the file as follows, where n stands for the value of the preceding replicator:

a) letter-k causes the current char number to be set to n ;
b) letter-x causes the char number to be incremented by n (10.5.1.2.o) ;
c) letter-y causes the char number to be decremented by n (10.5.1.2.p) ;
d) letter-l causes the line number to be incremented by n and the char number to be reset to one (10.5.1.2.q) ;
e) letter-p causes the page number to be incremented by n, and both the line number and the char number to be reset to one (10.5.1.2.r).

ii) A literal is written by writing the characters (strings) possessed by its constituent (row-of-)character-denotations each as many times as is the value of the preceding replicator; a string is written by writing its elements one after the other; a character is written by causing the character to be present at the current position of the file, thereby obliterating the character that was present, and then incrementing the char number by one.A literal is required by requiring the characters (strings) possessed by its constituent (row-of-)character-denotations, each as many times as is the value of the preceding replicator; a string is required by requiring its elements one after the other; a character is required by incrementing the char number by one if the character is present at the current position of the file; otherwise, the further elaboration is undefined.

jj) When a string whose number of characters is given is "read", then that number of characters are read and the result is a string whose elements are those characters; when a string is read under control of a given "terminator-string", then, as long as the line is not exhausted, characters are read up to but not including the first character which is the same as some element of the terminator-string, and the result is a string whose elements are those characters; when a character is read, then the result is the character present at the current position of the file, and the char number of the file is incremented by one.

kk) The mode specified by a picture is that enveloped by its pattern, if any. The number of characters specified by a picture is the sum of the numbers specified by its constituent frames and the number specified by a frame is equal to the value of its preceding replicator, if any, and one otherwise.

ll) On output, a picture may be used to "edit" a value in the following steps:

Step 1: The value is converted by an appropriate output routine (10.5.2.1.c,d,e) to a string of as many characters as specified by the picture (; if the pattern of the picture is an integral-pattern, then this conversion takes place to a base equal to the radix, if present, and base ten otherwise); if this number of characters is not sufficient, then value error of the file is called, a routine which may be provided by the programmer (10.5.1.kk);

Step 2: In those parts, if any, of the string specified by a sign-mould, a character specified by the sign-frame will be used to indicate the sign, viz., if the sign-frame is a minus-symbol and the value is positive, then a space, and, otherwise, the character specified by the sign-frame; this character is shifted in that part of the string specified by the sign-mould as far to the right as possible across all leading zeroes, and those zeroes are replaced by spaces (; e.g., under the sign-mould 4z+. the string possessed by "+0003" becomes that possessed by "...+3"); if the picture does not contain a sign-mould and the value is negative, then value error of the file is called;

Step 3: Leading zeroes in those parts of the string specified by any remaining zero-frames are replaced by spaces (; e.g., under the picture zdzd2d, the integer possessed by 180168 becomes the string possessed by "18.168";

Step 4: For all frames occurring in the picture, first the preceding insertion, if any, is performed, and next, if the frame is not "suppressed" (, i.e. preceded by letter-s), then that part of the string specified by the frame is written; finally, the insertion, if any, following the last constituent frame is performed.
(; e.g., editing under the picture zd"-"zd"-19"zd the integer possessed by 180168 causes the string possessed by 18-.1-1968 to be written).

5.5.1. continued 6

mm) On input a picture may be used to "indit" a value of a given mode from a file in the following steps:

Step 1: A string is obtained consisting of the characters obtained by performing the following process for all frames occurring in the picture, viz. first, the insertion, if any, preceding the frame is performed and next, as many characters are obtained as are specified by the frame; each of those characters is obtained,
if the frame is not suppressed, then by reading from the file a character, and, if the frame is a digit- (point-, exponent-, complex-) frame and the character is not a digit (point, ten to the power, plus i times), then calling char error of the file (10.5.1.kk) with as its parameter a zero (point, ten to the power, plus i times) and
if the frame is suppressed, then by taking, if the frame is a digit- (zero-, point-, exponent-, complex-, character-) frame a zero (zero, point, times ten to the power, plus i times);

Step 2: Those parts, if any, of the string specified by a sign-mould must contain a character, specified by the sign-frame, to indicate the sign (see ll Step 2); if those parts contain such a character, with only spaces appearing in front of it and no leading zeroes appearing after it, then those leading spaces, if any, are deleted; otherwise, disagreement of the file is called; if this character is a space, and the sign-frame is a minus-symbol, then it is replaced by a plus-symbol (; e.g., if in Step 1 under control of 3z-d, the string possessed by ". . . 39" is obtained, then in Step 2 that possessed by "+39" is obtained);

Step 3: Leading spaces in those parts of the string specified by any remaining zero-frames are replaced by zeroes;

Step 4: The string is converted by an appropriate input routine (10.5.2.2.c,d,e) into a value of the given mode, if possible, and, otherwise, value error of the file is called (; e.g., if max int(10.1.b) is 10000, then under +5d it is possible to input +10000, but not + 10001).

nn) A value of a given mode is compatible with a given picture if

a) on output, there exists some mode which is the mode specified by the picture preceded by zero or more times 'long', such that that mode is strongly coerced from the given mode;

b) on input, there exists some mode which is the mode specified by the picture preceded by 'reference-to' followed by zero or more times 'long', such that that mode is strongly coerced from the given mode.

(A value of mode 'reference-to-long-integral' is on output compatible with a picture that specifies the mode 'real', but not on input.))

oo) Formats have a complementary meaning on input and output, i.e. a given value which is not a string with one or two flexible bounds, which has been output successfully to the file, under control of a certain picture, starting from a certain position, can be successfully input again from that file under control of the same picture, starting at the same position, provided that the contents of the file are not changed in between; if the picture does not contain a letter-k or letter-y as alignment, and the picture does not contain any digit-frames or character-frames preceded by letter-s, then the second value, obtained on input, is equal (approxumately equal) to the given value if this is a string, integer or truth value (is a real value; output of this second value to the file has the same effect on the contents of the file as output of the given value under control of the same given picture and starting from one same position. }

5.5.2. Syntax of integral patterns

a) integral pattern{55e} : radix mould{b} option, sign mould{55e} option, integral mould{d} ; integral choice pattern{f}.

b) radix mould : insertion{55d} option, radix{c}, letter r{302b}.

c) radix{b} : digit two{303d} ; digit four{303d} ; digit eight{303d} ; digit one{303d}, digit zero{303d} ; digit one{303d}, digit six{303d}.

d) integral mould{a,553b,d,e} : loose replicatable suppressible digit frame{551m} sequence.

e) digit frame{55m} : zero frame{550} ; letter d{302b}.

f) integral choice pattern{a} : insertion{55d} option, letter c{302b}, literal{55j} list pack.

{Examples:

a) 2r6d30sd ; 12z+d ; zd"-"zd"-19"2d ;
120kc("mon","tues","wednes","thurs","fri","satur","sun") ;

b) 2r ;

c) 2 ; 4 ; 8 ; 10 ; 16 ;

d) zd"-"zd"-19"2d ;

f) 120kc("mon","tues","wednes","thurs","fri","satur","sun") }

{If a given value is transput under control of an integral pattern that begins with an integral-choice-pattern, then the insertion, if any, preceding the letter-c is performed, and,

a) on output, letting n stand for the integral value to be output, if n > 0 and the number of literals in the constituent literal-list-pack is at least n, then the n-th literal is written on the file; otherwise, the further elaboration is undefined;

b) on input, one of the constituent literals of the constituent literal-list-pack is required on the file; if the i-th constituent is the first one present, then the value is i; if none of these literals is present, then the further elaboration is undefined;

c) finally, the insertion, if any, following the pattern is performed;

otherwise, on output (input) the value is edited (indited) under control of the picture. }

5.5.3. Syntax of real patterns

a) real pattern{55c,556a} : sign mould{5511} option, real mould{b} ; floating point mould{d}.

b) real mould{a,e} : integral mould{5.5.2,d}, loose suppressible point frame{55m}, integral mould{552d}option ; loose suppressible point frame{55m}, integral mould{552d}.

c) point frame{55q} : point symbol{31b}.

d) floating point mould{a} : stagnant mould{e}, loose suppressible exponent frame{55m}, sign mould{551}option, integral mould{552d}.

e) stagnant mould{d} : sign mould{551} option, INTREAL mould{552d,553b}.

f) exponent frame{55q} : letter e{302b}.

{Examples:

a) +12d ; +d. 11de+2d ;

b) d.11d ; .12d ;

d) +d.11de+2d ;

e) +d.11d }

{If a value is transput under control of a real-pattern, then it is on output edited and on input indited under control of the picture. }

5.5.4. Syntax of boolean patterns

a) boolean pattern{55c} : insertion{55d} option, letter b{302b}, boolean choice mould{b} option.

b) boolean choice mould{a} : open symbol{31c}, literal{55g}, comma symbol{31e}, literal{55j}, close symbol{31e}.

{Examples:

a) 1"result"14xb ; b("","error") ;

b) ("", "error") }

{If the boolean-pattern does not contain a choice-mould, then the effect of using the pattern is the same as if the letter-b were followed by ("1","0").

The insertion, if any, preceding the letter-b is performed, and,

a) on output, if the truth value to be output is true, then the first constituent literal of the constituent choice-mould is written, and, otherwise, the second;

b) on input, one of the constituent literals of the constituent choice-mould is required on the file; if the first literal is present, then the value true is found; otherwise, if the second literal is present, then the value false is found; otherwise, the further elaboration is undefined;

c) finally, the insertion, if any, following the pattern is performed. }

5.5.5. Syntax of character patterns

a) character pattern{55c} : loose suppressible character frame{55m}.

b) character frame{55q} : letter a{302b}.

{Example:

a) "."a }

{If a given value is transput under control of a picture whose constituent pattern is a character pattern, then on output (input) the value is edited (indited) under control of the picture. }

5.5.6. Syntax of complex patterns

a) COMPLEX pattern{55c} : real pattern{553a}, loose suppressible
   complex frame{55m}, real pattern{553a}.
b) complex frame{55q} : letter b{302b}.
c)* complex pattern : COMPLEX pattern.

{Example:
a) 2x+.12de+2d"+j×"si+.10de+2d }

{If a given value is transput under control of a picture whose constituent pattern is a complex pattern, then on output (input) the value is edited (indited) under control of the picture. }

5.5.7. Syntax of string patterns

a) row of character pattern{55c} : loose string frame{55m} ;
   loose replicatable suppressible character frame{55m} sequence proper ;
   insertion{55d} option, replication{55g}, suppressible character
   frame{55q}.
b) string frame{55m} : letter t{302b}.

{Examples:
a) lt ; 5a3sa5a ; p"table_of"x10a }

{If a given value is transput under control of a picture whose pattern is a row-of-character-pattern, then, if the pattern is a loose-string-frame, then
a) the constituent insertion, if any, is performed;
b) on output, the given string is written on the file;
c) on input, if the string has fixed bounds, then that number of
   characters are read; otherwise, a string is read under control of the
   terminatorstring referenced by the file (10.5.1.mm);
d) finally, the insertion, if any, following the pattern is performed;
otherwise,
a) on output, the given string, which must have as many elements as the
   number of characters specified by the format-item, is edited;
b) on input, the string is indited. }

5.5.8. Transformats

a) structured with row of character field letter aleph letter aleph transformat{741b} : firm format unit{61e}.

{Example: (x≥0|f5df f5d"-"f)}

{Transformats are used exclusively as actual-parameters of formatted transput routines; for reasons of efficiency, the programmer has deliberately been made unable to use them elsewhere by the choice of letter aleph.

Although transformats are not denotations at all, they are handled here because of their close connection to formats. }

5.5.8.1. Semantics

a) The format {2.2.3.4} possessed by a given format-denotation is the same sequence of symbols as the given format-denotation.

b) A given transformat is elaborated in the following steps:

Step 1: It is preelaborated {1.1.6.f} ;

Step 2: A format-denotation is considered which is the same sequence of symbols as the format obtained in Step 1 ;

Step 3: All constituent dynamic-replications {5.5.1.h} of the considered format-denotation are elaborated collaterally {6.3.2.a}, where the elaboration of a dynamic-replication is that of its constituent serial-expression ;

Step 4: Each of those dynamic-replications is replaced by an integral-denotation {5.1.1} which possesses the same value as that dynamic-replication if that value is positive, and, otherwise, by a digit-zero; furthermore, every replicator which is empty is replaced by a digit-one ;

Step 5: That row-of-character-denotation {5.3} is considered which is obtained by replacing in the considered format-denotation as modified in Step 4 each constituent quote symbol by a quote image {5.3.1.d} and the first and the last constituent formatter-symbol by a quote-symbol ;

Step 6: A new instance of the value of the considered row-of-character-denotation is made to be the {only} field of a new instance of a structured value {2.2.3.2} whose mode is that obtained by deleting 'transformat' from that notion ending with 'transformat' of which the given transformat is a terminal production ;

Step 7: The transformat is made to possess the structured value obtained in Step 6.

6. Phrases

{A phrase is a declaration or a clause. Declarations may be unitary, e.g. *real* x, or collateral, e.g. *real* x, y. Clauses may be unitary, e.g. $x := 1$, collateral, e.g., $(x := 1, y := 2)$, closed, e.g. $(x + y)$ or conditional, e.g. *if* $x > 0$ *then* x *else* 0 *fi* (which may be written $(x > 0 \mid x \mid 0)$). Most clauses will be of a certain "sort", i.e. strong, weak, firm or soft, which determines how the coercions should be effected. The sort is "passed on" in the production rules for clauses and may be modified by "balancing" in serial-, collateral- and conditional-clauses. }

6.0.1. Syntax

a)* SOME phrase : SORTETY SOME PHRASE {61a,62a,b,c,d,f,63a,64a,c,d,e, 70a,81a}.

b)* SOME expression : SORTETY SOME MODE clause {61a,62b,c,d,f,63a, 64a,c,d,e,81a}.

c)* SOME statement : strong SOME void clause {61a,62b,63a,64a,c,e,81a}.

6.0.2. Semantics

a) The elaboration of a phrase begins when it is initiated, it may be "interrupted", "halted" or "resumed", and it ends by being terminated or completed, whereupon, if the phrase "appoints" a unitary-phrase as its "successor", then the elaboration of that unitary-phrase is initiated.

b) The elaboration of a phrase may be interrupted by an action {e.g. overflow} not specified by the phrase but taken by the computer if its limitations do not permit satisfactory elaboration. {Whether, after an interruption, the elaboration of the phrase is resumed, the elaboration of some unitary-phrase is initiated or the elaboration of the program ends, is left undefined in this Report. }

c) The elaboration of a phrase may be halted {10.4.a}, i.e. no further actions constituting the elaboration of that phrase take place until the elaboration of the phrase is resumed {10.4.b}, if at all.

d) A given clause is "protected " in the following steps:

Step 1: If the given clause contains a defining occurrence {4.1.2.a}(an indication-defining occurrence {4.2.2.a}) of a terminal production of a notion ending on 'identifier' ('indication') which also occurs outside it, then that defining (indication-defining) occurrence and all occurrences identifying it are replaced by occurrences of one same terminal production of that notion ~~'identifier' ('indication')~~ which does not occur in the program and Step 1 is taken; otherwise, Step 2 is taken;

Step 2: If the given clause contains an operator-defining occurrence {4.3.2.a} of a terminal production of a notion ending on 'indication' which also occurs outside it, then that operator-defining occurrence and all occurrences identifying it are replaced by occurrences of one same new terminal production of that notion ~~'indication'~~ which does not occur in the program and Step 3 is taken; otherwise, the protection of the given clause is complete;

Step 3: If the indication is a dyadic ~~priority~~-indication, then Step 4 is taken; otherwise, Step 2 is taken;

Step 4: A copy is made of the priority-declaration containing the indication which, before the replacement in Step 2, was identified by that operator-defining-occurrence; that indication in the copy is replaced by an occurrence of the new terminal production; the copy, thus modified, preceded by an open-symbol and followed by a go-on-symbol, is inserted preceding ~~by~~ the given clause, a close-symbol is inserted following the given clause, and Step 2 is taken.

{Clauses are protected in order to allow unhampered definitions of identifiers, indications and operators within ranges and to permit a meaningful call, within a range, of a procedure declared outside it. }

{What's in a name? that which we call a rose
By any other name would smell as sweet.
Romeo and Juliet, William Shakespeare. }

6.1. Serial clauses

{Serial-clauses are built from unitary-clauses and declarations with the help of go-on-symbols (*;*) and completion-symbols (. or *exit*), e.g., (*x* > *0* | *x*:= *1* | *l*); *y*. *l*: *y* + *1* , where the value of the clause is *y*, if *x* > *0* and *y* + *1* otherwise. A serial-clause may begin with a declaration-prelude, e.g., *int n*:= *1*; in *int n*:= *1*; *x*:= *y* + *n* . Labels may appear in only three syntactic positions within serial-clauses: after a completion-symbol (here a label is obligatory, e.g., .*l*:), in a sequencer (e.g., ;*l*:, or at the beginning of a clause-train (i.e. one or more unitary-clauses separated by sequencers, e.g., *l*: *x*:= *1*; *y*:= *2*). A declaration-prelude may contain constituent void-clauses (statements), but it does not begin or end with one, (e.g., [*1* : *n*]*real* 𝔛*1*; *for i to n do* 𝔛*1*[*i*]:= *i* × *i*; *real y*;), however, these void-clauses may not be labelled. A preface or a prelude always ends with a go-on-symbol. The modes of some serial-clauses must be balanced (6.1.1.g). For remarks concerning the balancing of modes see 6.4.1. }

6.1.1. Syntax

a) SORTETY serial CLAUSE {21d,55h,63a,64b,e} :
    declaration prelude{b} option,
    suite of SORTETY CLAUSE trains{f,g}.

b) declaration prelude{a,21b,c} : chain of declaration
    prefaces{c} separated by statement interlude{d} options.

c) declaration preface{b} :
    unitary declaration{70a}, go on symbol{31f} ;
    collateral declaration{62a}, go on symbol{31f}.

d) statement interlude{b} : chain of strong void units{e}
    separated by go on symbols{31f}, go on symbol{31f}.

e) SORTETY MOID unit{d,i,558a,62b,c,e,h,71t,74b,831f,861h,i} :
    SORTETY unitary MOID clause{81a}.

f) suite of STRONGETY CLAUSE trains{a,g} :
    chain of STRONGETY CLAUSE trains{h} separated by completers{l}.

g) suite of ~~FIRM~~ FEAT CLAUSE trains{a,g} : ~~FIRM~~ FEAT CLAUSE train{h} ;
    ~~FIRM~~ FEAT CLAUSE train{h}, completer{l},
    suite of strong CLAUSE trains{f} ;
    strong CLAUSE train{h}, completer{l},
    suite of ~~FIRM~~ FEAT CLAUSE trains{g}.

6.1.1. continued

h) SORTETY MOID clause train{f,g} : label{k} sequence option,
   statement prelude{i} option, SORTETY MOID unit{e}.

i) statement prelude{h} : chain of strong void units{e}
   separated by sequencers{j}, sequencer{j}.

j) sequencer{i} : go on symbol{31f}, label{k} sequence option.

k) label{h,j,l,21d} : label identifier{41b}, label symbol{31e}.

l) completer{f,g} : completion symbol{31f}, label{k}.

{Examples:

```
a) real a := 0 ; l1: l2: x := a + 1 ; (p | l3) ;
     (x > 0 | l3 | x := 1 - x) ; false. l3: y := y + 1 ; true ;
b) real a := 0 ; ;
c) real a := 0 ; ; int i, j ; ;
d) x := 0 ; (in real x ; x := 0 ; real y ;) ;
e) false ;
f) l1: l2: x := a + 1 ; (p | l3) ;
    (x > 0 | l3 | x := 1 - x) ; false. l3: y := y + 1 ; true ;
h) l1: l2: x := a + 1 ; (p | l3) ;
    (x > 0 | l3 | x := 1 - x) ; false ;
i) x := a + 1 ; (p | l3) ; (x > 0 | l3 | x := 1 - x) ;
j) ; l4: l5: ;
k) l4: ;
l) . l3: }
```

6.1.2. Semantics

a) The elaboration of a serial-clause is initiated by protecting it {6.0.2.d} and then initiating the elaboration of its textually first constituent unitary-phrase.

b) The completion of the elaboration of a unitary-phrase preceding a go-on-symbol initiates the elaboration of the textually first unitary-phrase after that go-on-symbol.

c) The elaboration of a serial-clause is

- interrupted (halted, resumed) upon the interruption (halting, resumption) of a constituent unitary-phrase ;
- terminated upon the termination of the elaboration of a constituent unitary-phrase appointing a successor outside the serial-clause, and that successor {8.2.7.2.b.Step 2} is appointed the successor of the serial-clause.

6.1.2. continued

d) The elaboration of a serial-clause is completed upon the completion of the elaboration of its textually last constituent unitary-clause or of that of a constituent unitary-clause preceding a completer.

e) The value of a serial-clause is the value of that constituent unitary-clause the completion of whose elaboration completed the elaboration of the serial-clause provided that the scope {2.2.4.2} of that value is larger than the serial-clause {; otherwise, the value of the serial-clause is undefined}.

{In *y := (x := 1.2 ; 3.4)*, the value of the serial-clause *x := 1.2 ; 3.4* is the real number possessed by *3.4*. In *xx := (real r := 0.1 ; r)*, the value of the serial-clause *real r := 0.1 ; r* is undefined since the scope of the name possessed by *r* is the serial-clause itself, whereas, in *y := (real r := 0.1 ; r)*, the serial-clause *real r := 0.1 ; r* possesses a real value.}

6.2. Collateral phrases

{Collateral-phrases contain two or more unitary-phrases separated by comma-symbols (*,* or *comma*) and, in the case of collateral-clauses, are enclosed between an open-symbol (*(* or *begin*) and a close-symbol (*(* or *end*), e.g. *(x := 1, y := 2)* or *real x, real y* (usually *real x, y*, see 9.2.c). The values of collateral-clauses which are not statements (void-clauses) are either of multiple or of structured mode, e.g. *(1.2, 3.4)* in *[]real x1 = (1.2, 3.4)* and in *compl z := (1.2, 3.4)*. Here, the collateral-clause *(1.2, 3.4)* acquires the mode 'row of real' or the mode 'COMPLEX'. Collateral-clauses whose value is structured must contain at least two fields, for, otherwise, in the range of *struct m = (ref m m) ; m nobuo, yoneda*, the assignation *nobuo := (yoneda)* would be ambiguous. In the range of *struct r = (real a) ; r r*, the construction *r := (3.14)* is not an assignation, but *a of r := 3.14* is. It is possible to present a single value or no value at all as a multiple value, e.g. *[]real x1 := ; []real y1 := 3*, but this involves a coercion known as *rowing*, see 8.2.6.}

6.2.1. Syntax

a) collateral declaration{61c} : unitary declaration{70a} list proper.

6.2.1. continued

b) strong collateral void clause{81d} : parallel symbol{31e} option,
   strong void unit{61e} list proper pack.

c) strong collateral row of MODE clause{81d} :
   parallel symbol{31e} option,
   strong MODE unit{61e} list proper pack.

d) FEAT collateral row of MODE clause{81d} :
   parallel symbol{31e} option, FEAT MODE balance{e} pack.

e) FEAT MODE balance{c} : FEAT MODE unit{61e},
   comma symbol{31b}, strong MODE unit{61e} list ;
   strong MODE unit{61e}, comma symbol{31b}, FEAT MODE unit{61e}.
   strong MODE unit{61e}, comma symbol{31b}, FEAT MODE balance{e}.

f) strong collateral structured with FIELDS and FIELD clause{81d} :
   parallel symbol{31e} option,
   strong structured with FIELDS and FIELD structure{g} pack.

g) strong structured with FIELDS and FIELD structure{f,g} :
   strong structured with FIELDS structure{g,h}, comma symbol{31b},
   strong structured with FIELD structure{h}.

h) strong structured with MODE named TAG structure{g} :
   strong MODE unit{61e}.

{Examples:

a) real *x*, real *y* ; (and by 9.2.c) real *x*, *y* ;

b) *(x := 1, y := 2, z := 3)* ;

c) *(x, n)* ;

d) *(1.2, 3, 4)* (in *(1.2, 3, 4) + x1*, supposing + has been declared also
   for 'row of real') ;

e) *(1.2, 3, 4)* (in *(1.2, 3, 4) + x1*); *(1, 2.3)* (in *(1, 2.3) + x1*) ;
   (1, 2.3, 4) (in *(1, 2.3, 4) + x1*) ;

f) *(1, 2.3)* (in *z := (1, 2.3)*) ;

g) *1, 2.3* ;

h) *1* }

6.2.2. Semantics

a) If constituents of an occurrence of a ~~members of constituents of a given~~ terminal production of a notion are "elaborated collaterally", then this elaboration is the collateral action {2.2.5} consisting of the {merged} elaborations of these constituents, and is

initiated by initiating the elaboration of each of these constituents,
interrupted upon the interruption of the elaboration of any of these constituents,
completed upon the completion of the elaboration of all of these constituents, and
terminated upon the termination of the elaboration of any of these constituents, and if that constituent appoints a successor, then this is the successor of the given terminal production.

b) A collateral-declaration is elaborated by elaborating its constituent unitary-declarations collaterally {a}.

c) A collateral-clause is elaborated in the following steps:
Step 1: Its constituent units are elaborated collaterally {a} ;
Step 2: If the notion generating {1.1.6.c.vii} the collateral-clause envelopes {1.1.6.j} a mode, then this mode is considered and Step 2 is taken; otherwise,{it envelopes 'void' and} the elaboration of the collateral-clause is complete;
Step 3: If the considered mode begins with 'row of', then Step 4

is taken; otherwise, new instances of the values obtained in Step 1 are made, in the given order, to be the fields of a new instance of a structured value {2.2.3.2.} ; this structured value is considered and Step 6 is taken ;
Step 4: If the values of the units obtained in Step 1 are names {2.2.3.5} one or more of which refers to an element or subvalue having one or more states {2.2.3.3 } equal to zero, or if the values of these units are multiple values, not all of whose corresponding upper (lower) bounds are equal, then the further elaboration is undefined; otherwise, Step 5 is taken ;
Step 5: A new instance of a multiple value,
is created as follows:
let "m" stand for the number of constituent units in the collateral-clause ;
if the values obtained in Step 1 are not multiple values,
then its element with index "i" is a new instance of the value of the i-th constituent unit and its descriptor consists of an offset 1 and one quintuple (1,m,1,1,1) ;

otherwise, those values are multiple values and the elements with indices $(i - 1) \times r + j$, $j = 1, \ldots, r$ of the new value, where r stands for the number of elements in one of those values, are new instances of the elements of the value of the i-th constituent unit and the descriptor of the new value is a copy of the descriptor of the value of one of the constituent units into which an additional quintuple (1,m,1,1,1) has been inserted in front of the old first quintuple, the offset has been set to 1, d_n has been set to 1, and, for i = n, n-1, ... ,2, the stride d_{i-1} has been set to $(u_i - l_i + 1) \times d_i$;
this new multiple value is considered and Step 6 is taken ;

Step 6: The value of the collateral-clause is the considered value; its ~~mode is that obtained in Step 2;~~ mode is the considered mode.

6.3. Closed clauses

{Closed-clauses are generally used to construct primaries (8.1.1.d) from serial-clauses, e.g. *(x + y)* in *(x + y) × a*. The question of identification (Chapter 4) and protection (6.0.2.d) may arise in closed-clauses, because a serial-clause is a range (4.1.1.e) and it may begin with a declaration-prelude (6.1.1.a). }

6.3.1. Syntax

a) SORTETY closed CLAUSE{81d} : SORTETY serial CLAUSE{61a} pack.

{Examples:

a) <u>*begin*</u> *i* := *i* + *1* ; *j* := *j* + *1* <u>*end*</u> ; *(x + y)* ;

6.3.2. Semantics

The elaboration of a closed-clause is that of its serial-clause, and its value is that, if any, of its serial-clause.

6.4. Conditional clauses

{Conditional-clauses allow the programmer to choose one out of a pair of clauses, depending on the value (which is of mode 'boolean') of a condition, e.g. *(x > 0 | x | 0)*. Here, *x > 0* is the condition. If the condition is true, then the value is *x*; otherwise, it is *0*. Conditional-clauses are generalized in the extensions 9.4.a,b,c, e.g. *if x > 0 then x elsf x < -1 then -(x + 1) else 0 fi*, which has the same effect as *(x > 0 | x |(x < -1 | -(x + 1)| 0))*. Unlike similar constructions in other languages, conditional-clauses are always enclosed between an if-symbol, represented by *if* or by *(*, and a fi-symbol represented by *fi* or by *)*. This enclosure allows both parts of the choice-clause and the condition to contain serial-clauses. }

6.4.1. Syntax

a) SORTETY conditional CLAUSE{81d} : if symbol{31e},
   condition{b}, SORTETY choice CLAUSE{c,d}, fi symbol{31e}.

b) condition{a} : strong serial boolean clause{61a}.

c) STRONGETY choice CLAUSE{a} :
   STRONGETY then CLAUSE{e}, STRONGETY else CLAUSE{e} option.

d) FEAT choice CLAUSE{a} :
   FEAT then CLAUSE{e}, strong else CLAUSE{e} option ;
   strong then CLAUSE{e}, FEAT else CLAUSE{e}.

e) SORTETY THELSE CLAUSE{c,d} :
   THELSE symbol{31e}, SORTETY serial CLAUSE{61a}.

{Examples:

a) *(x > 0 | x | 0)* ; *if overflow then exit fi* ;

b) *x > 0* ; *overflow* ;

c) *| x | 0* ; *then exit* ;

d) *(x > 0 | x | 0)* (in *(x > 0 | x | 0) + y)* ;

e) *|x* ; *| 0* ; *then exit* }

{Rule d illustrates the necessity for the "balancing" of modes (see also 6.1.1.g). Thus, if a choice-clause is, say, firm, then at least one of its two constituent clauses must be firm, while the other may be strong. For example, in *(p | x | skip) + (p | skip | y)*, the conditional-clause *(p | x | skip)* is balanced by making *| x* firm and *| skip* strong, whereas *(p | skip | y)* is balanced by making *| skip* strong and *| y* firm. The example *(p | skip | skip) + y* illustrates that not both may be strong, for otherwise the operator + could not be identified. }

6.4.2. Semantics

a) A conditional-clause is elaborated in the following steps:

Step 1: Its condition is elaborated ;

Step 2: If the value of that condition is true, then the then-clause and otherwise the else-clause, if any, of its choice-clause is considered ;

Step 3: The serial-clause of the considered clause, if any, is elaborated ;

Step 4: The value, if any, of the conditional-clause, then is that of the clause elaborated in Step 3, if any.

b) The elaboration of a conditional-clause is

- interrupted (halted, resumed) upon the interruption (halting, resumption) of the elaboration of the condition or the considered clause ;
- completed upon the completion of the elaboration of the considered clause, if any; otherwise, completed upon the completion of the elaboration of the condition ;
- terminated upon the termination of the elaboration of the condition or considered clause, and, if one of these appoints a successor, then this is the successor of the conditional-clause.

7. Unitary declarations

{Unitary-declarations provide the indication-defining occurrences of mode-indications, e.g. string in mode string = [1:flex]char and ~~prior~~ dyadic-indications, e.g. plus in priority plus = 1, ~~the~~ defining occurrences of mode-identifiers, e.g. x in real x, and the operator-defining occurrences of operators, e.g. abs in op abs = (int a) int : (a < 0 | -a | a). Declarations occur in declaration-preludes (6.1.1.b).}

7.0.1. Syntax

a) unitary declaration{61c,62a} : mode declaration{72a} ;
   priority declaration{73a} ; identity declaration{74a} ;
   operation declaration{75a}.

{Examples:

a) mode string = [1:flex]char ; priority plus = 1 ;
   int m = 4096 ; op ÷ = (real a, b)int : round a ÷ round b }

7.0.2. Semantics

A mode-identifier (operator) which was caused to possess a value by the elaboration of a declaration containing the defining (operator-defining) occurrence of that mode-identifier (operator) is caused to possess an undefined value upon termination or completion of the elaboration of the smallest range {4.1.1.e} containing that declaration.

7.1. Declarers

{Declarers are built from the symbols int, real, bool, char, format, with the assistance of such symbols as long, ref, [,], struct, union and proc. A declarer specifies a mode, e.g. real specifies the mode 'real'. A declarer is either a declarator or a mode-indication, e.g. compl is a mode-indication and not a declarator. Declarers are classified as actual, formal or virtual depending on the kind of lower- and upper-bounds which are permitted. Formal declarers have the greatest freedom in this respect,

7.1. continued

e.g., [1:n]real, [1:flex]real, [1:either]real and []real, ~~are~~ may be all formal, but only the first two ~~are~~ may be actual and only the last ~~is~~ may be virtual.}

7.1.1. Syntax

a)* declarer : VICTAL MODE declarer{b}.

b) VICTAL MODE declarer{h,l,m,n,o,w,y,z,54c,d,g,72a,851b,c} :
 VICTAL MODE declarator{c,d,e,k,l,m,n,o,v,y} ;
 MODE mode indication{42b}.

c) VICTAL PRIMITIVE declarator{b,d} : PRIMITIVE symbol{31d}.

d) VICTAL long INTREAL declarator{b,d} :
 long symbol{31d}, VICTAL INTREAL declarator{c,d}.

{Examples:

b) real ; bits ;

c) int ; real ; bool ; char ; format ;

d) long int ; long long real }

e) VICTAL structured with FIELDS declarator{b} :
 structure symbol{31d}, VICTAL FIELDS declarator{f,h,k}pack.

f) VICTAL FIELDS and FIELD declarator{e,f,k} :
 VICTAL FIELDS declarator{f,h,i,k}, comma symbol{31e}, VICTAL FIELD declarator{h,k}.

g)* field declarator : VICTAL FIELD declarator{h,k}.

h) VICTAL STOWED field TAG declarator{e,f} :
 VICTAL STOWED declarer{b}, STOWED field TAG selector{j}.

i)* field selector : FIELD selector{j}.

j) MODE named TAG selector {h,852a} : TAG{302b,41c,d}.

k) VICTAL NONSTOWED field TAG declarator{e,f}:
 virtual NONSTOWED declarer{b}, NONSTOWED field TAG selector{j}.

{Examples:

e) struct (string title, [1:n]ref string pages, int price) ;

f) string title, [1:n] ref string pages, int price ;

h) [1:n] ref string pages ;

j) title ;

k) int price }

{Rule hand k, together with 1.2.1.r,s,t,u,v and 4.1.1.c,d leads to an infinity of production rules of the strict language, thereby enabling the syntax to "transfer" the field-selectors (i) into the mode of structured values, and making it ungrammatical to use an "unknown" field-selector in a selection (8.5.2). Concerning the occurrence of a given field-selector more than once in a declarer, see 4.4.3, which implies that struct(real x, int x) is not a (correct) declarer, whereas struct(real x, struct(int x, bool p) p) is. Notice, however, that the use of a given field-selector in two different declarers within a given reach does not cause ambiguity. Thus, mode cell = struct(string name, ref cell next) and mode link = struct(ref link next, ref cell value) may both occur in the same reach. }

l) VIRACT reference to MODE declarator{b} :
   reference to symbol{31d}, virtual MODE declarer{b}.
m) formal reference to STOWED declarator{b} :
   reference to symbol{31d}, formal STOWED declarer{b}.
n) formal reference to NONSTOWED declarator{b} :
   reference to symbol{31d}, virtual NONSTOWED declarator{b}.

{Examples:
l) ref[ ]real ;
m) ref[1:]real ; ref[1:either, 1:flex]real ;
n) ref ref[ ]real }

{Rules l, m and n imply that, for instance, ref[1:either]real x may be a formal-parameter (5.4.1.f), whereas ref ref[1:either]real x may not.}

o) VICTAL ROWS structured with FIELDS declarator{b} :
   sub symbol{31e}, VICTAL ROWrower {9,r}, bus symbol{31e},
   VICTAL structured with FIELDS declarer{b}.
p) VICTAL ROWS NONSTOWED declarator :
   sub symbol{31e}, VICTAL ROWS rower {9,r}, bus symbol{31e},
   virtual NONSTOWED declarer{b}.
q) VICTAL row of ROWS rower{o,p} :
   VICTAL row of rower{q}, comma symbol{31e}, VICTAL ROWS rower{p,q}.
r) VICTAL row of rower{o,p} :
   VICTAL lower bound{r,s,u}, up to symbol{31e}, VICTAL upper BOUND{r,s,u}.
s) virtual LOWPER bound{q} : EMPTY.
t) actual LOWPER bound{q}: strict LOWPER bound{t} ;
   strict LOWPER bound{t} option, flexible symbol{31d}.

7.1.1. continued 3

u) strict LOWPER bound{s,u,861f} : strong integral unit{61e}.

v) formal LOWPER bound{q} :
   strict LOWPER bound{t} option, flexible symbol{31d} option;
   strict LOWPER bound{t} option, either symbol{31d}.

{Examples :

o) [1:m] struct ([1:n] real a, int b) ;

p) [1:m,1:n] ref [] real ;

q) 1:m,1:n ;

r) 1:m ;

t) m ; m flex ; flex ;

u) m ;

v) m flex ; either}

⊥ contained in a formal-declarer
⊤ the corresponding

{The flexible-symbol, either-symbol, strict-lower-bound and strict-upper-bound⊥ serve to prescribe states and bounds of the multiple value~~s~~ possessed by ⊤actual-parameter~~s~~. The flexible-symbol in ref [1:flex]char s = t prescribes that a name referring to a multiple value with upper state 0 (i.e. the upper bound may vary) will be possessed by s; the either-symbol in ref[1:n either]char s = t prescribes that that upper state is either 0 or 1 (i.e. the upper bound may be variable or fixed) and the absence of both flexible-symbol and either-symbol in ref[1:n]char s = t prescribes that that upper state is 1 (i.e. the upper bound must be fixed). Independently, n in ref[1:n either]char s = t or in ref[1:n]char s = t prescribes that a name referring to a multiple value whose upper bound equals the value of n will be possessed by s; if, in the first example, the upper state is 0, then that upper bound may well be changed later on by an assignation. The absence of a strict-upper-bound in ref[1:flex] char s = t does not restrict the upper bound in that way. Similar remarks apply to strict-lower-bounds. The flexible-symbol, strict-lower-bound and strict-upper-bound serve a similar role in generators (8.5)}

w) VICTAL PROCEDURE declarator{b} :
   procedure symbol{31d}, virtual PROCEDURE plan{54c,d}.

x) virtual MODE parameter{54c,e} : virtual MODE declarer{b}.

y)* parameters pack : VICTAL PARAMETERS{w,54e,f,74b} pack.

{Examples:

w) proc ; proc(real, int) ; proc(real)bool ;

x) real }

z) VICTAL union of MOODS and MOOD mode declarator {b} :
union of symbol {31d}, MOODS and MOOD and open box {aa} pack.

aa) LMOODSETY MOOD and open BOX {z, bb, cc} :
LMOODSETY closed MOOD end BOX {bb, cc}.

bb) LMOODSETY closed LOSETY MOOD end box {aa, bb, cc} :
LMOODSETY closed LOSETY MOOD and MOOD end box {bb};
LMOODSETY open LOSETY MOOD and box {aa, dd}.

cc) LMOODSETY closed LOSETY MOOD end MOOT and BOX {aa, cc} :
LMOODSETY closed LOSETY MOOT and MOOD end BOX {bb, cc};
LMOODSETY closed LOSETY MOOD and MOOD end MOOT and BOX {cc};
LMOODSETY open LOSETY MOOD and MOOT and BOX {aa, dd}.

dd) open MOODS and MOOD BOX {bb, cc, ff} :
open MOODS and box {dd, ee}, comma symbol {31e}, MOOD BOX {ff, gg}.

ee) open MOOD and box {dd} : MOOD box {gg}.

ff) MOODS and MOOD box {dd} :
union of symbol {31d}, open MOODS and MOOD box {dd} pack;
union of MOODS and MOOD mode indication {42b}.

gg) MOOD box {dd, ee} : virtual MOOD declarer {b}.

{Examples:

z) union (real, union (int, bool), union (real, int));
union (ri, union (bool, real)) (in the reach of union ri = (real, int)).

Let "b" stand for 'boolean', "i" for 'integral', "r" for 'real', "A" for 'and' and "[bir]" for one of the six protonotions 'bAiAr', 'bArAi', 'iAbAr', 'iArAb', 'rAbAi' and 'rAiAb'. Both examples are of a virtual-, actual- or formal- union-of-[bir]-mode-declarator. The choice ofor [bir] is left undefined and is semantically irrelevant, but if one chooses some canonical ordering of all modes involved in a program, then the rules z upto gg and 8.2.4.1.a, b, c, d do not cause any ambiguity (see 1.1.6.i). If "B" stands for 'box', "C" for 'closed', "E" for 'end', "K" for 'comma symbol' and 'O' for 'open' then the production

of the first example from 'actual union of integral and real and boolean mode declarator' is suggested by
(z): iArAbAOB :(aa): iArACbEB :(bb): iArAObAB :(aa): iACrEbAB :
:(cc): iACrArEbAB :(cc): iACrAbArEB :(bb): iAOrAbArAB
:(aa): CiErAbArAB :(cc): CrAiEbArAB :(cc): CrAiAiEbArAB
:(cc): CrAiAbAiErAB :(cc): CrAiAbArAiEB :(bb): OrAiAbArAiAB
:(dd): OrAiAbAB, K, rAiB :(dd): OrAB, K, iAbB, K, rAiB
:(ee): rB, K, iAbB, K, rAiB.

7.1.2. Semantics

a) A given declarer specifies that mode which is obtained by deleting 'declarer' and the terminal production of the metanotion 'VICTAL' from that direct production {1.1.2.c} of the notion 'declarer' of which the given declarer is a production.

b) A given declarer is developed as follows:
Step: If it is, or contains, a mode-indication which is an actual declarer or formal-declarer, then that indication is replaced by a copy of the actual-declarer of that mode-declaration {7.2} which contains its indication-defining occurrence {4.2.2.b}, and the Step is taken; otherwise, the development of the declarer has been accomplished. }

{A declarer is developed during the elaboration of an actual-declarer (c) or identity-declaration (7.4.2.Step 1).}

c) A given actual-declarer is elaborated in the following steps:
Step 1: It is developed {b} ;
Step 2: If it now begins with a structure symbol, then Step 4 is taken; otherwise, if it now begins with a sub-symbol, then Step 5 is taken; otherwise, if it now begins with a union-of-symbol, then Step 3 is taken; otherwise, a new instance of a value of the mode specified {a} by the given actual-declarer is considered and Step 8 is taken ;

Step 3: Some mode is considered which does not begin with 'union of' and from which the mode specified by the given actual-declarer is united {4.4.3.a}, a new instance of a value whose scope is the program and which is of the considered mode is considered and Step 8 is taken ;

Step 4: All its constituent actual-declarers are elaborated collaterally {6.3.2.a}; the values referred to by the values {names} of these actual-declarers are made, in the given order, to be the fields of a new instance of a structured value of the mode specified by the given actual-declarer; this structured value is considered, and Step 8 is taken ;

Step 5: All its constituent strict-lower-bounds and strict-upper-bounds are elaborated collaterally ;

Step 6: A descriptor {2.2.3.3} is established consisting of an offset 1 and as many quintuples, say "n", as there are constituent actual-row-of-rowers in the given declarer; if the i-th of these actual-row-of-rowers contains a strict-lower-bound (strict-upper-bound), then l_i (u_i) is set equal to its value; otherwise, l_i (u_i) is undefined; if the i-th of these actual-row-of-rowers contains an actual-lower-bound (actual-upper-bound) which is or contains the flexible-symbol, then s_i (t_i) is set to 0; otherwise, s_i (t_i) is set to 1; next d_n is set to 1, and, for i = n, n-1, ... , 2, the stride d_{i-1} is set to $(u_i - l_i + 1) \times d_i$;

Step 7: The descriptor is made to be the descriptor of a multiple value of the mode specified by the given actual-declarer; ~~each of~~ its elements are obtained as follows : if the last constituent declarer of the given actual-declarer is an actual-declarer, then it is elaborated a number of times and each element is a new instance of the value referred to by one of the resulting names ; otherwise, each element is a new instance of some value of some mode {not beginning with 'union of' and} such that the mode specified by the last constituent virtual-declarer is or is united from {4.4.3.a} it; this multiple value is considered ;

Step 8: A name {2.2.3.5} different from all other names and whose mode is 'reference to' followed by the mode specified by the actual-declarer, is created and made to refer to the considered value; this name is the value of the given actual-declarer.

7.2. Mode declarations

{Mode declarations provide the indication-defining occurrences of mode-indications, which act as abbreviations for declarers built from primitive symbols, e.g. mode string = [1:flex]char, or struct(string title, ref book next).

from other declarers or even from themselves, e.g. mode book =

7.2. continued

In this last example, the mode-indication is not only a convenient abbreviation but it is essential to the declaration. }

7.2.1. Syntax

a) mode declaration{70a} : mode symbol{31d},
    MODE mode indication{42b}, equals symbol{31c},
    actual MODE declarer{71b}.

{Examples:

a) mode string = [1:flex]char ;
    struct compl = (real re, im) (see 9.2.b,c) ;
    union primitive = (int, real, book, char, format) (see 9.2.b) }

7.2.2. Semantics

The elaboration of a mode-declaration involves no action.
{See 4.4.4.c concerning certain mode-declarations, e.g. mode a = a, which are not contained in proper programs.}

7.3. Priority declarations

{Priority-declarations provide the indication-defining occurrences of dyadic-indications, e.g. o in priority o = 6, which may then be used in the declaration of dyadic operations. Priorities from 1 to 9 are available. Since monadic-operators have effectively only one priority level (8.4.1.g), which is higher than that of all dyadic-operators, they do not appear in priority-declarations.}

7.3.1. Syntax

a) priority declaration{70a} : priority symbol{31d},
    priority NUMBER indication{42e}, equals symbol{31c},
    NUMBER token{b,c,d,e,f,g,h,i,j}.

b) one token{a} : digit one symbol{31b}.

c) TWO token{a} : digit two symbol{31b}.

d) THREE token{a} : digit three symbol{31b}.

7.3.1. continued

e) FOUR token{a} : digit four symbol{31b}.
f) FIVE token{a} : digit five symbol{31b}.
g) SIX token{a} : digit six symbol{31b}.
h) SEVEN token{a} : digit seven symbol{31b}.
i) EIGHT token{a} : digit eight symbol{31b}.
j) NINE token{a} : digit nine symbol{31b}.

{Example:
a) priority + = 6 }

7.3.2. Semantics

The elaboration of a priority-declaration involves no action. {For a summary of the standard priority-declarations, see the remarks in 8.4.2.}

7.4. Identity-declarations

{Identity-declarations provide ~~the~~ defining occurrences of mode-identifiers, e.g. x in real x (which is an abbreviation of ref real x = loc real, see 9.2.a). Their elaboration causes mode-identifiers to possess values; ~~in the~~ here, ~~example~~ x is made to possess a name which refers to some real value.}

7.4.1. Syntax

a) identity declaration{70a} : formal MODE parameter{54g},
   equals symbol{31c}, actual MODE parameter{b}.
b) actual MODE parameter{a,54c,e,75a,862a} :
   strong MODE unit{61e} ; MODE local generator{851b,-} ;
   MODE local assignation{831b,-} ; MODE transformat{558a,-} .

{Examples:

```
a) real e = 2.718281828459045 ; int e = abs i ;
   real d = re(z × conj z) ; ref[,]real a1 = a[,:k] ;
   ref real x1k = x1[k] ; compl unit = 1 ;
   proc int time = clock ÷ cycles ;
```

7.4.1. continued

(The following declarations are given first without, and then with, the extensions of 9.2)

```
     ref real x = loc real ; real x ;
     ref int sum = loc int := 0 ; int sum := 0 ;
     ref [,]real a = loc[1:m,1:n]real := x2 ; [1:m,1:n]real a := x2 ;
     proc(real)real vers = ((real x)real : 1 - cos(x)) ;
       proc vers = (real x)real :  1 - cos(x) ;
     ref proc(real)real p = loc proc(real)real ;
       proc(real)real p ;
     ref proc(real)real q = loc proc(real)real :=
                              ((real x)real : (x > 0 | x | 1)) ;
       proc q := (real x)real : (x > 0 | x | 1) ;
b)   abs i ; loc real ; loc int := 0 ; f+d.11de+2df }
```

7.4.2. Semantics

An identity-declaration is elaborated in the following steps:

Step 1: The formal-declarer of its formal-parameter is developed {7.1.2.b} ;

Step 2: Its actual-parameter and all ~~[illegible]~~ strict-lower-bounds and strict-upper-bounds ~~of~~ contained in that formal-declarer, as possibly modified in Step 1, are elaborated collaterally {6.3.2.a}, ~~and~~ if the value of the actual-parameter is a name, then the value to which that name refers, or otherwise the value itself, is considered ;

Step 3: If the considered value is an element or subvalue of a multiple value {2.2.3.3} having one or more states equal to zero, then the further elaboration is undefined; otherwise, ~~Step 4 is taken;~~

Step 4: If the considered value is not a multiple value, then Step 7 is taken; otherwise, if the value of the actual-parameter is not a name, then Step 6 is taken ;

Step 5: For each ~~[illegible]~~ flexible-symbol-option ~~of~~ contained in the formal-declarer, as possibly modified in Step 1, {the corresponding state is checked, i.e.} if that flexible-symbol-option is the flexible-symbol (empty) and the corresponding state in the considered value is 1(0), then the further elaboration is undefined; otherwise, Step 6 is taken ;

Step 6: For each ~~[illegible]~~ strict-lower-bound and strict-upper-bound ~~of~~ contained in but not contained in any strict-lower-bound or strict-upper-bound contained in it, the formal-declarer, as possibly modified in Step 1, but not contained in any strict-lower-bound or strict-upper-bound contained in it, {the corresponding bound is checked, i.e.} if its value is not the same as the corresponding bound in the considered value, then the further elaboration is undefined; otherwise, Step 7 is taken ;

Step 7: The identifier of the formal-parameter is made to possess ~~a new instance of~~ the value of the actual-parameter.

{According to Step 6, the elaboration of the declaration [1:2]real x1 = (1.2,3.4,5.6) is undefined and according to Step 5 the elaboration of the declaration ref[1:flex]real x1 = [1:2]real := (1.2,3.4) is undefined. The elaboration of the declaration [1:flex]real x1 = (1.2,3.4) is well defined but its effect is also obtained by the elaboration of the less confusing declaration []real x1 = (1.2,3.4). }

7.5. Operation declarations

{Operation-declarations provide the operator-defining occurrences of operators, e.g., op ∨ =(real a, b)real : (random < .5 | a | b), which contains an operator-defining occurrence of ∨ as a dyadic-operator. Unlike identity-declarations of which no two for the same identifier may occur in a reach (4.4.2.b), more than one operation-declaration involving the same adic-indication may occur in the same reach, see 10.2.2.i, 10.2.3.i, etc.}

7.5.1. Syntax

a) operation declaration{70a} : PRAM caption{b},
   equals symbol{31c}, actual PRAM parameter{74b}.

b) PRAM caption{a} : operation symbol{31d},
   virtual PRAM plan{54c}, PRAM ADIC operator{43b,c}.

{Examples:

a) op ∧ = (bool a, b)bool : (a | b | false) ;
   op abs = (real a)real : (a < 0 | -a | a) (see 9.2.d,e).

b) op(bool, bool)bool ∧ ; op(real)real abs }

7.5.2. Semantics

An operation-declaration is elaborated in the following steps:

Step 1: Its actual-parameter is elaborated ;

Step 2: The operator of its caption is made to possess the {routine which is the} value obtained in Step 1.

{The formula (8.4.1) p ∧ q, where ∧ identifies the operator-defining occurrence of ∧ in the operation-declaration

op ∧ = (bool john, proc bool mccarthy)bool : (john | mccarthy | false),

possesses the same value as it would if ∧ identified the operator-defining occurrence of ∧ in the operation-declaration

op ∧ = (bool a, b)bool : (a | b | false),

except, possibly, when the elaboration of q involves side effects on that of p.}

8. Unitary clauses

{Unitary-clauses may occur as actual-parameters, e.g. x in sin(x), as sources in assignations, e.g. y in x := y, as strict-lower (upper)-bounds, new-lower (upper)-bounds or subscripts, e.g. m, 0 and n in x2[:m <u>at</u> 0,n], as bodies in routine-denotations, e.g. i <u>plus</u> 1 in ((<u>ref</u> <u>int</u> i)<u>int</u> : i <u>plus</u> 1), or may be used to construct serial- or collateral-clauses, e.g. x := 1 in (x := 1 ; y := 2) or in (x := 1, y := 2). Unitary-clauses either are closed, collateral or conditional, or are "coercends". There are four kinds of coercends: confrontations, e.g. x := 1, formulas, e.g. x + 1, cohesions, e.g. next <u>of</u> cell, and bases, e.g. x. These coercends and the closed-, collateral- and conditional-clauses are grouped into the following four classes, each class being a subclass of the next: primaries, which may be subscripted and parametrized, e.g. x1 and sin in x1[i] and sin(x); secondaries, from which fields may be selected, e.g. z in re <u>of</u> z, and tertiaries, which may be operands, or may be destinations in assignations, or may occur in identity- or conformity-relations, e.g. x in x + 1 or in x := 1 or in x :=: yy or in x ::= ir, and, finally, unitary-clauses, which is the largest class. Thus, r <u>of</u> s(i) means that s is first called or subscripted and a field is then selected, while (r <u>of</u> s)(i) means that the field is selected first. Also, r <u>of</u> s + t means that the field is selected from s before elaborating the routine possessed by +, while to force the elaboration of + first, one must write r <u>of</u> (s + t). }

8.1.1. Syntax

a) SORTETY unitary MOID clause{61e} :
    SORTETY MOID tertiary{b} ;
    SORTETY MOID confrontation{820d,e,f,g,830a,-}.
b) SORTETY MOID tertiary{a,831e,832a,833a} :
    SORTETY MOID secondary{c} ;
    SORTETY MOID ADIC formula{820d,e,f,g,84b,g}.
c) SORTETY MOID secondary{b,84f,852a} :
    SORTETY MOID primary{d} ;
    SORTETY MOID cohesion{820d,e,f,g,850a}.

8.1.1. continued

d) SORTETY MOID primary{c,861a,862a} :
    SORTETY CLOSED MOID clause{62b,c,d,f,63a,64a,-} ;
    SORTETY MOID base{820d,e,f,g,860a}.

{Examples:
a) x ; x := 1 ;
b) x ; x + 1 ;
c) x ; **real** ;
d) (x + 1) ; x }

8.2. Coercends

{Coercends are of four kinds: bases, e.g. x, cohesions, e.g. re **of** z, formulas, e.g. x + y and confrontations, e.g. x := 1. These notions are collectively considered as coercends because it is in their production rules that the basic coercions occur.

In current programming languages certain implicit changes of type are described, usually in the semantics. Thus x := 1 may mean that the integral value of 1 yields an equivalent real value which is then assigned to x. In ALGOL 68, such implicit changes of mode are known as coercions, and are reflected in the syntax. Certain coercions available in other languages, such as i := x, are not permitted. One must write i := **round** x or i := **entier** x, for in this situation it is felt advisable for the programmer to state the coercion explicitly. Apart from this, all the coercions which the programmer might reasonably expect, are supplied.

There are eight basic coercions. They are: dereferencing, deproceduring, proceduring, uniting, widening, rowing, hipping and voiding. In x + 3.14, the base x, whose a priori mode is 'reference to real', is dereferenced to 'real'; in x := random, the base random, whose a priori mode is 'procedure real', is deprocedured to 'real'; in **proc** p = **go to** north berwick, the jump, **go to** north berwick, which has no a priori mode, is procedured to 'procedure void'; in **union**(**int**, **real**) ir := 1, the base 1, whose a priori mode is 'integral', is united to 'union of integral and real mode'; in x := 1 the base 1, whose a priori mode is 'integral', is widened to 'real'; in **string** s := 'a', the base 'a', whose a priori mode is 'character', is rowed to 'row of character'; in x := **skip**, the skip **skip**, which has no a priori mode, is hipped to 'real' and in (x := 1 ; y := 2) the confrontation x := 1, whose a priori mode is 'reference to real', is voided (i.e. its value is ignored).

The kinds of coercion which are used depend upon three things: "syntactic position", a priori mode and a posteriori mode (i.e. the modes before and after coercion). There are four sorts of syntactic positions. They are: "strong" positions, i.e. actual-parameters, e.g. x in sin(x), sources, e.g. x in y := x, conditions, e.g. x > 0 in (x > 0 | x | 0), subscripts, e.g. i in x1[i] etc.; "firm" positions, i.e. operands, e.g. x in x + y, transformats, e.g. £5d£, and certain primaries, e.g. sin in sin(x); "weak" positions, e.g. certain primaries, e.g. x1 in x1[i] and certain secondaries, e.g. z in re of z; and "soft" positions, i.e. destinations, e.g. x in x := y, certain other tertiaries, e.g. xx in xx :=: x, and monadic-operands in depressions, e.g. xx in val xx.

Strong positions are so called because the a posteriori mode is dictated entirely by the context. Such positions lead to the possibility of any of the eight basic coercions. Firm positions are e.g. operands, in which widening, rowing, hipping and voiding must be excluded, since, otherwise, the identification of the operations involved in i + j, x + y (supposing + to be declared also for 'row of real'), i + skip and i + algol could not be properly made. In the weak positions, only deproceduring and dereferencing are permitted, and special care must be taken that dereferencing ~~looks ahead and does not~~ removes a 'reference to' ~~which precedes~~ only if followed by 'reference to'. ~~[illegible] mode.~~ The x1 in x1[i] := 1 demonstrates the necessity for this look-ahead. In the soft positions, the a posteriori mode is the a priori mode except for the removal of zero or more 'procedure's. Thus in soft positions only deproceduring is performed.

In the productions of a notion, the sort (strong, firm, weak, soft) of position is passed on, or modified during balancing (to strong) and leads to basic coercions which appear in the production rules for coercends; moreover, the coercion must be completely expended in these rules. For example, y in x := y is a real-source and therefore a strong-real-unit (8.3.1.1.f); the sort 'strong' is passed through the productions of 'strong real unit' until a 'strong real base' is reached (8.1.1.d); this is then produced to 'strongly dereferenced to real base' (8.2.0.1.d), next to 'reference to real base' (8.2.1.1.a) and finally to 'reference to real identifier' (8.6.0.1.a). }

8.2.0.1. Syntax

a)* coercend : SORT COERCEND{d,e,f,g,830a,84b,g,850a,860a,-} ;
    SORTly ADAPTED to COERCEND{821a,b,822a,b,c,823a,824a,825a,b,c,d,826a,
    827a,828a,b,-}.

b)* SORT coercend : SORT COERCEND{d,e,f,g}.

c)* SORTly ADAPTED coercend : SORTly ADAPTED to COERCEND.

d) strong COERCEND{81a,b,c,d} :
    COERCEND{830a,84b,g,850a,860a,-} ;
    strongly ADAPTED to COERCEND{821a,822a,823a,824a,825a,b,c,d,826a,827a,
    828a,b,-}.

e) firm COERCEND{81a,b,c,d,84d,f} : COERCEND{830a,84b,g,850a,860a,-} ;
    firmly ADJUSTED to COERCEND{821a,822a,823a,824a,-}.

f) weak COERCEND{81a,b,c,d} : COERCEND{830a,84b,g,850a,860a,-} ;
    weakly FITTED to COERCEND{821b,822b}.

g) soft COERCEND{81a,b,c,d,84f} : COERCEND{830a,84b,g,850a,860a} ;
    softly deprocedured to COERCEND{822c}.

{Examples:

d) 3.14 (in x := 3.14) ; y (in x := y) ;

e) 3.14 ; x (in 3.14 + x) ; sin (in sin(x)) ;

f) x1 (in x1[i]) ; zz(in re of zz in the reach of ref compl zz;);

g) x (in x := 1) ; xory (in xory := 3.14) }

8.2.1. Dereferenced coercends

{Coercends are dereferenced when it is required that an initial 'reference to' should be removed from the a priori mode; e.g. in x := y, the a priori mode of y is 'reference to real' but the a posteriori mode required in this strong position is 'real'. Here y possesses a name which refers to a real value and it is the real value which is assigned to x, not the name. }

8.2.1.1. Syntax

a) STIRMly dereferenced to MODE FORM{a,820d,e,822a,823a,824a,825a,b,826a} :
   reference to MODE FORM{830a,84b,g,850a,860a} ;
   STIRMly FITTED to reference to MODE FORM{a,822a}.

b) weakly dereferenced to reference to MODE FORM{b,820f} :
   reference to reference to MODE FORM{830a,84b,g,850a,860a} ;
   weakly FITTED to reference to reference to MODE FORM{b,822b}.

{Examples:

a) y (in x := y or in x + y) ; yy (in x := yy or in x + yy) ;
b) ~~x1 (in x1[i]) ;~~ rx1 (in rx1[i] in the reach of ref[~~[illegible]~~]real rx1;) }

8.2.1.2. Semantics

A dereferenced-coercend is elaborated in the following steps:
Step 1: It is preelaborated {1.1.6.i} ;
Step 2: If the value obtained in Step 1 is not nil, then the value of the dereferenced-coercend is a ~~[illegible]~~ copy of the value referred to by the name obtained in Step 1 {;otherwise, the further elaboration is undefined}.

{Weak dereferencing must look ahead so that it does not remove a 'reference to' which precedes a mode which is 'NONREF'. For example, in x1[i] := y, the primary x1 should not be dereferenced, for x1[i] must be a name. In x1[i]+y, the x1 is not dereferenced but the base x1[i] is. }

8.2.2. Deprocedured coercends

{Coercends are deprocedured when it is required that an initial 'procedure' should be removed from the a priori mode; e.g. in x := random, the a priori mode of random is 'procedure real' but the a posteriori mode required in this strong position is 'real'. Here the routine possessed by random is elaborated and the real value yielded is assigned to x. }

8.2.2.1. Syntax

a) STIRMly deprocedured to MOID FORESE{a,820d,e,821a,824a,825a,b,826a,828b} :
   procedure MOID FORESE{84b,g,850a,860a} ;
   STIRMly FITTED to procedure MOID FORESE{a,821a}.

8.2.2.1. continued

b) weakly deprocedured to MODE FORESE{820f,821b} :
    procedure MODE FORESE{84b,g,850a,860a} ;
    firmly FITTED to procedure MODE FORESE{a,821a}.

c) softly deprocedured to MODE FORESE{c,820g} :
    procedure MODE FORESE{84b,g,850a,860a} ;
    softly deprocedured to procedure MODE FORESE{c}.

{Examples:

a) random (in x := random or in x + random) ;

b) rz (in re <u>of</u> rz in the reach of <u>proc</u> rz = <u>compl</u> : (random,random)) ;

c) xory (in xory := 1) }

8.2.2.2. Semantics

A deprocedured-coercend is elaborated in the following steps:

Step 1: It is preelaborated {1.1.6.i} and a copy is made of {the routine which is} the resulting value ;

Step 2: The deprocedured-coercend is replaced by the copy obtained in Step 1, and the elaboration of the copy is initiated; if this elaboration is completed or terminated, then the copy is replaced by the deprocedured-coercend before the elaboration of a successor is initiated.
{See also calls, 8.6.2.}

8.2.3. Procedured coercends

{Coercends are procedured when it is required that an initial 'procedure' should be placed before the a priori mode (i.e. they should be turned into procedures without parameters), e.g. x := 1 in <u>proc</u> <u>real</u> p := x := 1. However, special care must be taken with procedures which deliver no value, in order that clauses like (<u>proc</u> p, q ; p := q := stop) should not be ambiguous. Here the routine possessed by stop is assigned to q and then to p, but is not elaborated. In (<u>proc</u> p ; p := x := 1) however, 1 is not assigned to x, but that routine which assigns 1 to x is assigned to p. The relevant syntax is described by the productions of rule 8.2.3.1.b.}

8.2.3.1. Syntax

a) STIRMly procedured to procedure MOID FORM{a,820d,e,824a,826a} :
   MOID FORM{830a,84b,g,850a,860a,-} ;
   STIRMly dereferenced to MOID FORM{821a,-} ;
   STIRMly procedured to MOID FORM{a,-} ;
   STIRMly united to MOID FORM{824a,-} ;
   STIRMly widened to MOID FORM{825a,b,-} ;
   STIRMly arrayed to MOID FORM{826a,-} ;
   STIRMly provisional MOID FORM{b,-}.

b) strongly provisional void monadic formula {a}:
   void symbol {31e}, SOFT NONPROC monadic operand {84f}.

{Examples:

a) 3.14 (in proc real p := 3.14) ; x (in proc real p = x) ;
   3.14 (in proc proc real := 3.14) ;
   1 (in proc union(int, real) p := 1) ;
   1 (in proc real p := 1) ; 1 (in proc[]int p := 1) ;
   3.14 (in proc p := 3.14) ;

b) void (x := 1) (in proc p := void (x := 1))}

8.2.3.2. Semantics

A procedured-coercend is elaborated in the following steps:

Step 1: A copy is made of it {itself, not its value}⊥and an open-symbol followed by a routine-symbol is placed before and a close-symbol is placed after the copy ;

Step 2: The mode obtained by deleting 'ly procedured to' and the terminal productions of 'STIRM' and 'FORM' from that notion as terminal production of which the procedured-coercend is elaborated, is considered; if this considered mode is not 'procedure void' then the initial 'procedure' is deleted from the considered mode and a virtual-declarer specifying the mode so obtained is inserted between the open-symbol and the routine-symbol in the copy ;

Step 3: The routine possessed by the routine-denotation {5.4.2} obtained in Step 2 is the value of the procedured-coercend.

⊥ , the initial void-symbol, if any, of the copy is deleted

8.2.3.2. continued

{The elaboration of (real : (p | x | -x)) yields the routine (val (real := (p | x | -x))), whereas that of the strong-conditional-procedure-real-clause (p | x | -x) yields either the routine (val (real := x)) or the routine (val (real := -x)), depending on the value of p. Similarly, the elaboration of (real:(x := x + 1 ; y)) yields the routine (val (real := (x := x + 1 ; y))), whereas that of the strong-closed-procedure-real-clause (x := x + 1 ; y) yields, apart from a change in the value of x, the routine (val (real := y)).}

8.2.4. United coercends

{Coercends are united when it is required that the a priori mode should be changed to a mode united from (4.4.3.a) it, e.g. in union(int, real) ir := 2, the base 2 is of a priori mode 'integral' but the source of this assignation requires the mode 'union of integral and real mode'.}

8.2.4.1. Syntax

a) STIRMly united to union of MOOD and MOODS mode FORM{820d,e,823a,826a}:
one out of MOOD and MOODS mode FORM {b};
of and MOOD and MOODS not union of MOOD mode FORM {c}.

b) one out of LMOODSETY MOOD RMOODSETY mode FORM{a}:
MOOD FORM{830a,84b,g,850a,860a};
firmly FITTED to MOOD FORM {821a,822a};
firmly procedured to MOOD FORM{823a,-}.

c) of LMOODSETY and MOOD RMOODSETY not union of MOODS mode FORM{a}:
of LMOODSETY MOOD and RMOODSETY not union of MOODS mode FORM {c,d};
of LMOODSETY RMOODSETY not union of MOODS and MOOD mode FORM {c,d}.

d) of MOOD and MOODS and not UNITED FORM {c}:
union of MOOD and MOODS mode FORM {84b,g,850a,860a,-};
firmly FITTED to union of MOOD and MOODS mode FORM {821a,822a}.

{Examples:

a) 2 ; ir ;

b) 2 ; i ; **true** ;

d) ri ; ir   (all in (**union**(**int**, **real**) **ir** ; **ir** ir ; **ir** ri = (p | j | x);
union (**ir**, **proc** **bool**) irb := 2; irb := i; irb := **true**))}

{In uniting, 'strong' leads to 'firm' in order that unions like that involved in **union**(**int**, **real**) ir := 1 should not cause ambiguities. In this example, if the base 1 is widened it cannot then be united, i.e. in the order of productions in the syntax, uniting cannot be followed by widening.}

8.2.5. Widened coercends

{Coercends are widened when it is required that the a priori mode should be changed from 'integral' to 'real' or from 'real' to 'COMPLEX', e.g. 1 in z := 1, or from 'BITS' to 'row of boolean', or from 'BYTES' to 'row of character'. }

8.2.5.1. Syntax

a) strongly widened to LONGSETY real FORM{b,820d,823a,826a} :
   LONGSETY integral FORM{830a,84b,g,850a,860a} ;
   strongly FITTED to LONGSETY integral FORM{821a,822a}.

b) strongly widened to structured with REAL named letter r letter e
   and REAL named letter i letter m FORM{820d,823a,826a} :
   REAL FORM{830a,84b,g,850a,860a} ;
   strongly FITTED to REAL FORM{821a,822a} ;
   strongly widened to REAL FORM{a}.

c) strongly widened to row of boolean FORM{820d,823a,826a} :
   BITS FORM{830a,84b,g,850a,860a} ;
   strongly FITTED to BITS FORM{821a,822a}.

d) strongly widened to row of character FORM{820d,823a,826a} :
   BYTES FORM{830a,84b,g,850a,860a} ;
   strongly FITTED to BYTES FORM{821a,822a}.

{Examples:

a) 1 (in x := 1) ; i (in x := i) ;

b) 3.14 (in z := 3.14) ; x (in z := x) ; 1 (in z := 1)

c) 1 0 1 ; t (in [1 : 3] bool b1 := (p | 1 0 1 | t) ;

d) ctb "abc" ; r (in s := (p | ctb "abc" | r)) }

8.2.5.2. Semantics

A widened-coercend is elaborated in the following steps:

Step 1: It is preelaborated {1.1.6.i} and the value yielded is considered;

Step 2: If the considered value is an integer, then the value of the widened-coercend is a new instance of that real number which is equivalent to that integer {2.2.3.1.d}; otherwise, if the considered value is a real number, then the value of the widened-coercend is a new instance of that structured {complex (10.2.5)} value composed

8.2.6. Rowed coercends

{Coercends are rowed when it is required that 'row of' should be placed either before the a priori mode or after an initial 'reference to' of the a priori mode; e.g., in [1:1] real a1 := 3.14, the a priori mode of the base 3.14 is 'real' but the a posteriori mode required in this strong position is 'row of real', whereas, in ref [1:] real a2 = x, the a priori mode of the base x is 'reference to real' but the a posteriori mode required is 'reference to row of real'. Here, the value of 3.14, to which x refers, is turned into a multiple value with a descriptor. Note that the value of a2[1] :=: x is true.}

8.2.6.1. Syntax

a) strongly rowed to REFETY row of MODE FORM {a, 820d, 823a}:
   REFETY MODE FORM {830a, 84b,g, 850a, 860a};
   strongly ADJUSTED to REFETY MODE FORM {821a, 822a, 823a, 824a, -};
   strongly widened to REFETY MODE FORM {825a, b, c, d, -};
   strongly rowed to REFETY MODE FORM {a, -};
   REFETY MODE FORM vacuum {b, -}.

b) REFETY NONROW base vacuum {a}: EMPTY.

{Examples:

a) 3.14 (in [1:1] real x1 := 3.14); y (in ref [1:1] real x1 = y);
   3.14 (in [1:1] proc real p := 3.14;
   3.14 (in [1:1] compl z1 := 3.14);
   3.14 (in [1:1,1:1] real x2 := 3.14; y (in ref [1:1,1:1] real x2 = y);
   (the EMPTY following := in [1:0] real :=)}

8.2.6.2. Semantics

A rowed-coercend is elaborated in the following steps:

Step 1: The mode obtained by deleting 'strongly rowed to' and the terminal production of 'FORM' from that notion as occurrence of a terminal production of which the rowed-coercend is elaborated is considered; if the rowed-coercend is not empty, then it is preelaborated, the value obtained and its scope are considered and Step 3 is taken;

Step 2: A new instance of a multiple value {2.2.3.3} composed of zero elements and a descriptor consisting of an offset 1 and one quintuple (1,0,1,1,1) is considered, and Step 7 is taken;

Step 3: If the considered mode does not begin with 'reference to', then Step 5 is taken; otherwise, if the considered value is not nil, then Step 4 is taken; otherwise, the elaboration of the rowed-coercend is complete, its value is a new instance of nil whose mode is the considered mode;

Step 4: That instance of the value to which the {name which is the} considered value refers is considered instead; if the considered value is a multiple value having one or more states equal to zero,

8.2.6.2. continued

or if it is an element or subvalue of such a multiple value, then the further elaboration is undefined; otherwise, Step 5 is taken;

Step 5: If the considered value is a multiple value, then Step 6 is taken; otherwise, a new instance of a multiple value composed of the considered value as only element and of a descriptor consisting of an offset 1 and one quintuple (1,1,1,1,1) is considered instead, and Step 7 is taken;

Step 6: A new instance of a new multiple value, composed of the elements of the considered value and a descriptor which is a copy of the descriptor of the considered value into which the additional quintuple (1,1,1,1,1) {the value of the stride is irrelevant} is inserted before the first quintuple, and in which all states have been set to 1, is considered instead;

Step 7: If the considered mode does not begin with 'reference to', then the value of the rowed-coercend is the considered value; otherwise, a name different from all other names, whose scope is the considered scope; {i.e., considered in Step 1} and whose mode is the considered mode, is created; this name is made to refer to the considered value and is the value of the rowed-coercend.

8.2.7. Hipped coercends

{Coercends are hipped when they are skips, jumps or nihils. Though there is no a priori mode, whatever mode is required by the context, is adopted, e.g., in real x = skip, the base, skip, which has no a priori mode, is hipped to 'real'. Since hipped coercends are so very accommodating, no other coercions may follow them (in the elaboration order); otherwise, ambiguities might appear. Consider, for example, the several meanings of union(int, real, bool, char) u := skip, supposing uniting could follow hipping. }

8.2.7.1. Syntax

a) strongly hipped to MOID base{820d} : MOID hop{b} ; MOID nihil{e,-}.
b) MOID hop{a} : skip{c} ; jump{d}.
c) skip{b} : skip symbol{31g}.

8.2.7.1. continued

d) jump{b} : go to symbol{31f} option, label identifier{41b}.
e) reference to MODE nihil{a} : nil symbol{31g}.

{Examples:
a) skip ; nil ;
b) skip ; go to grenoble ;
c) skip ;
d) go to grenoble ; st pierre de chartreuse ;
e) nil }

8.2.7.2. Semantics

a) The value of a skip is a new instance of some value whose mode is that obtained in the following steps:

Step 1: The mode obtained by deleting 'hop' from that notion ending with 'hop' of which the skip is an occurrence of a terminal production is considered ;

Step 2: If the considered mode begins with 'union of', then some mode which does not begin with 'union of' and from which the considered mode is united {~~2.2.4.1.b~~ 4.4.3.a} is considered instead; the considered mode is the mode of the value of the skip.

b) A jump is elaborated in the following steps:

Step 1: The mode obtained by deleting 'hop' from that notion ending with 'hop' of which the jump is an occurrence of a terminal production is considered ;

Step 2: If the considered mode does not begin with 'procedure', then the elaboration of the unitary-clause which is the jump is terminated and it appoints as its successor the first unitary-clause textually after the defining occurrence {in a label (4.1.2)} of the label-identifier occurring in the jump; otherwise, Step 3 is taken ;

Step 3: A copy is made of the jump and an open-symbol followed by a routine-symbol is placed before and a close-symbol is placed after the copy; if the considered mode is not 'procedure void', then the initial 'procedure' is deleted from the considered mode and a virtual declarer specifying the mode so obtained is inserted between the open-symbol and the routine-symbol in the copy; the value of the jump is that of the routine-denotation consisting of the same sequence of symbols as the copy.

c) The elaboration of a nihil involves no action; its value is a new instance of nil {2.2.3.5.a} whose mode is that obtained by deleting

8.2.7.2. continued

'nihil' from that notion ending with 'nihil' of which the nihil is an occurrence of a terminal production.

{Skips play a role in the semantics of routine-denotations (5.4.2. Step 2) and calls (8.6.2.2.Step 4). Moreover, they are useful in a number of programming situations, like e.g.;

i) supplying an actual-parameter (7.4.1.b) whose value is irrelevant or is to be calculated later; e.g. f(3, skip) where f does not use its second actual-parameter if the value of the first actual-parameter is positive; see also 11.11.ax ;

ii) supplying a constituent unit of a collateral-clause (6.2.1.b,c,e,h), e.g. [1 : 4]real x1 := (3.14, skip, 1.68, skip) ;

iii) as a dummy statement (6.0.1.c) in those rare situations where the use of a completer is inappropriate, e.g. l: skip) in 10.4.a.

A jump is useful as a clause to terminate the elaboration of another clause when certain requirements are not met, e.g. go to exit in y := if x ≥ 0 then sqrt(x) else go to exit fi, or f in (j > a | f | j) from 10.2.3.r.

If e1, e2 and e3 are label-identifiers, then the reader might recognize the effect of the declaration []proc switch = (e1,e2,e3) and the statement switch[i]; however, the declaration [1 : flex]proc switch := (e1,e2,e3) is perhaps more powerful, since assignations like switch[2] := e1 and switch := (e1,e2,e3,e4) are possible.

A nihil is useful particularly where structured values are connected to one another in that a field of each structured value refers to another one except for one or more structured values where the field does not refer to anything at all; such a field must then be nil. }

8.2.8. Voided coercends

{Coercends are voided when it is required that their values (and therefore modes) should be ignored, e.g. in (x := 1 ; y := 2), the confrontation x := 1, whose a priori mode is 'reference to real', is voided (see 6.1.1.i). Confrontations must be treated differently from the other coercends in order that, e.g. in (proc p ; p := stop ; p), the confrontation p := stop does not involve the elaboration of stop, but in the last occurrence of p, the routine possessed by stop is elaborated.}

8.2.8.1. Syntax

a) strongly voided to void confrontation{820d} : MODE confrontation{830a}.
b) strongly voided to void FORESE{820d,h} :
   NONPROC FORESE{84b,g,850a,860a} ;
   strongly deprocedured to NONPROC FORESE{822a}.

{Examples:
a) x := 1 (in (x := 1 ; y := 2)) ;
b) x ; random (in (x ; random ; skip)) }

{The value obtained by elaborating (i.e. preelaborating 1.1.6.i) a voided-coercend is discarded.}

{In the reach of the declaration []proc switch = (e1,e2,e3) and the clause-train e1:e2:e3:stop, the construction switch ; stop is not a serial-clause because switch is not a strong-void-unit. In fact, switch cannot be deprocedured, because its mode begins with 'row of' and no coercion will remove the 'row of' and it cannot be 'voided' because 'row of procedure void' is not a terminal production of 'NONPROC'. However, the elaboration of switch[2] ; skip will involve a jump to the label e2. }

8.3. Confrontations

8.3.0.1. Syntax

a) MODE confrontation{81a,820d,e,f,g,821a,b,823a,b,824a,825a,b,c,d,826a,828a} :
   MODE nonlocal assignation{831b,-};
   MODE conformity relation{832a,-} ;
   MODE identity relation{833a,-}.

{Examples:
a) x := 3.14 ; ec :: e (see 11.11.q) ; xx :=: xory }

8.3.1. Assignations

{In assignations, e.g. x := 3.14, a value is assigned to a name. In x := 3.14, the value possessed by the source 3.14 is assigned to the value (name) possessed by x. A distinction must be made between nonlocal-assignations which

8.3.1. continued

are unitary-clauses, and local-assignations which are not. A local-assignation is an actual-parameter and thus may be used to initialize a declaration, e.g. loc real := 3.14, which is contained in real x := 3.14, before the extension of 9.2.a is made. }

8.3.1.1. Syntax

a)* assignation : MODE LOCAL assignation{b,-}.
b) reference to MODE LOCAL assignation{830a,74b} :
  reference to MODE LOCAL destination{d,e},
  becomes symbol{31c}, MODE source{f}.
c)* destination : MODE LOCAL destination{d,e,-}.
d) reference to MODE local destination{b} :
  reference to MODE local generator{851b}.
e) reference to MODE nonlocal destination{b} :
  soft reference to MODE tertiary{81b}.
f) MODE source{b} : strong MODE unit{61e}.

{Examples:
b) x := 1 ; loc real := 3.14 ;
d) loc real ;
e) x ;
f) 1 ; 3.14 }

8.3.1.2. Semantics

a) When a given instance of a value is superseded by another instance of a value, then the name which refers to the given instance is caused to refer to that other instance, and, moreover, each name which refers to an instance of a multiple or structured value of which the given instance is a component {2.2.2.k} is caused to refer to the instance of the multiple or structured value which is established by replacing that component by that other instance.

b) When an element (a field) of a given multiple (structured) value is superseded by another instance of a value, then the mode of the thereby established multiple (structured) value is that of the given value.

8.3.1.2. continued

c) An instance of a ~~A~~ value is assigned to a name in the following steps:

Step 1: If the given value does not refer to an element or subvalue of a multiple value having one or more states equal to zero {2.2.3.3.b}, if the scope of the given name is not larger than the scope of the given value {2.2.4.2} and if the given name is not nil, then Step 2 is taken; {otherwise, the further elaboration is undefined;}

Step 2: The instance of the value referred to by the given name is considered; if the mode of the given name does not begin with 'reference to union of' and the considered ~~value~~ instance is one of a multiple value or a structured value, then Step 3 is taken; otherwise, the considered ~~value~~ instance is superseded {a} by a ~~new instance~~ copy of the given ~~value~~ instance and the assignment has been accomplished ;

Step 3: If the considered value is a structured value, then Step 5 is taken; otherwise, applying the notation of 2.2.3.3.b to its descriptor, for i = 1, ... , n, if $s_i = 0$ ($t_i = 0$), then l_i (u_i) is set to the value of the i-th lower bound (i-th upper bound) in the descriptor of the given value; moreover, for i = n, n-1, ... , 2, the stride, d_{i-1}, is set to $(u_i - l_i + 1) \times d_i$; finally, if some $s_i = 0$ or $t_i = 0$, then the descriptor of the considered value, as modified above, is made to be the descriptor of a new instance of a multiple value which is of the same mode as the considered value, and this new instance is made to be referred to by the given name and is considered instead ;

Step 4: If for all i, i = 1, ... , n, the bound l_i (u_i) in the descriptor of the considered value, as possibly modified in Step 3, is equal to l_i (u_i) in the descriptor of the given value, then Step 5 is taken {; otherwise, the further elaboration is undefined} ;

Step 5: Each field (element, if any) of the given value is assigned {in an order which is left undefined} to the name referring to the corresponding field (element, if any) of the considered value and the assignment has been accomplished.

d) An assignation is elaborated in the following steps:

Step 1: Its destination and source are elaborated collaterally {6.2.2.a} ;

Step 2: The value of its source is assigned to the value {name} of its destination ;

Step 3: The value of the assignation is ~~a new instance of~~ the value of its destination.

{Observe that (x, y) := (1.2, 3.4) is not an assignation, since (x, y) is not a destination; the mode of the value of a collateral-clause (6.2.1.c,d,f) does not begin with 'reference to' but with 'row of' or 'structured with'. }

8.3.2. Conformity relations

{The purpose of conformity-relations is to enable the programmer to find out the current mode of an instance of a value if the context only restricts this mode to be one of a number of given modes. See for example 11.11.q,r,s,ak,al,am. Conformity relations are thus used in conjunction with unions. }

{I would to God they would either conform, or be more wise, and not be catched!
Diary, 7 Aug. 1664, Samuel Pepys.}

8.3.2.1. Syntax

a) boolean conformity relation{830a} :
  soft reference to LMODE tertiary{81b},
  conformity relator{b}, RMODE tertiary{81b}.

b) conformity relator{a} : conforms to symbol{31c} ;
  conforms to and becomes symbol{31c}.

{Examples:

a) ec :: e (see 11.11.q) ; ev ::= e (see 11.11.r) ;

b) :: ; ::= }

8.3.2.2. Semantics

A conformity-relation is elaborated in the following steps:

Step 1: Its tertiaries are elaborated collaterally {6.2.2.a} and the value of its textually last tertiary is considered ;

Step 2: If the mode of the value of its textually first tertiary is 'reference to' followed by a mode which is or is united from {4.4.3.a} the mode of the considered value, then the value of the conformity-relation is true and Step 4 is taken; otherwise, Step 3 is taken ;

Step 3: If the considered value refers to another value, then this other value is considered instead and Step 2 is taken; otherwise, the value of the conformity-relation is false and Step 4 is taken ;

Step 4: If its conformity-relator is a conforms-to-and-becomes-symbol and the value of the conformity-relation is true, then the considered value is assigned {8.3.1.2.c} to the value of the textually first tertiary.

{Although not suggested by the wording of Step 2, the, possibly, most obvious applications of conformity-relations are those in which 'RMODE' (8.3.2.1.a) begins with 'union of' whereas 'LMODE' does not. Then, the mode of the considered value (Step 1) is not 'RMODE' (which is united from it) and the conformity-relation serves to ask whether this mode is 'LMODE' and, if so and if the conformity-relator is a conforms-to-and-becomes-symbol, to assign this value to a name whose mode does not begin with 'reference to union of' and, thereby, make this value easily available elsewhere. (See, e.g., 11.q,r,s,z,aa,ak,al,am,ar).

Observe that if the considered value is an integer and the mode of its textually first tertiary is 'reference to' followed by a mode which is or is united from the mode 'real' but not from 'integral', then the value of the conformity-relation is false. Thus, ~~in contrast with assignation,~~ no automatic widening from 'integral' to 'real' takes place. For example, in union(real, bool) rb ; rb ::= 1, no value is assigned to rb, but in rb ::= 1.0 and in, rb := 1.0, the assignment/ takes place. Rule 8.3.2.1.b is the only rule in the syntax which allows the production of uncoerced clauses, i.e. those produced from 'RMODE tertiary'. }

8.3.3. Identity-relations

{Identity-relations may be used in order to ask whether two names of the same mode are the same ; e.g., in the reach of the declarations struct cons = (ref cong car, cdr); union cong = (cons, string); cons cell := (cong := "abc", nil), the identity-relation cdr of cell :=: nil possesses the value false because the value of cdr of cell is the name referring to the second field of the structured value referred to by the value of cell and, hence, is not nil, but the value of val cdr of cell :=: nil possesses the value true .}

8.3.3.1. Syntax

a) boolean identity relation{830a} :
   soft reference to MODE tertiary{81b}, identity relator{b},
   strong reference to MODE tertiary{81b} ;
   strong reference to MODE tertiary{81b}, identity relator{b},
   soft reference to MODE tertiary{81b}.

b) identity relator{a} : is symbol{31c} ; is not symbol{31c}.

{Examples:

a) xory :=: x ; xx :=: x ;

b) :=: ; :≠: }

8.3.3.2. Semantics

An identity-relation is elaborated in the following steps:

Step 1: Its tertiaries are elaborated collaterally{6.2.2.a} ;

Step 2: If its identity-relator is an is-symbol (is-not-symbol) then the value of the identity-relation is true (false) if the values {names} obtained in Step 1 are the same and false (true) otherwise.

{Assuming the assignations xx := yy := x, the value of the identity-relation xx :=: yy is false because xx and yy, though of the same mode, do not possess the same name {7.1.2.Step 8}, but the name which each possesses refers to the same name and so val xx :=: val yy possesses the value true. The value of the identity-relation xx :=: xory has a 1/2 probability of being true because the value possessed by xx (effectively val xx here, because of coercion) is the name possessed by x, and the routine possessed by xory (see 1.3), when elaborated, yields either the name possessed by x or, with equal probability, the name possessed by y. In the identity-relation, the programmer is usually asking a specific question concerning names and thus the level of reference is of crucial importance. Thus at least one of the tertiaries of an identity-relation must be soft, i.e. must involve only deproceduring and certainly no dereferencing. The construction case i in x, xx, xory, nil esac :=: case j in y, skip, xory, re of z,yy esac is an example of a delicately balanced identity-relation in which the mode is 'reference to real'.

Observe that the value of the formula 1 = 2 is false, whereas 1 :=: 2 is not an identity-relation, since the values of its tertiaries are not names. Also <u>f2d3df</u> :=: <u>f5df</u> is not an identity-relation, whereas <u>f2d3df</u> = <u>f5df</u> is a formula, but involves an operation which is not included in the standard-prelude.}

8.4. Formulas

{Formulas are either dyadic, e.g. x + i, or monadic, e.g. <u>abs</u> x. A formula has at least one operand an at least one operant, i.e. operator or dereference-symbol. The order of elaboration of a formula is determined by the priority of its operators; monadic formulas are elaborated first and then the dyadic formulas from the highest to the lowest priority. Since the dereference-symbol is not an operator, the programmer is prevented from changing its meaning. }

8.4.1. Syntax

a)* SORTETY formula : SORTETY MOID ADIC formula{b,g,820d,e,f,g}.

b) MOID PRIORITY formula{81b,820d,e,f,g,821a,b,822a,b,c,823a,b,824a,825a, b,c,d,826a,828b} : firm LMODE PRIORITY operand{d}, procedure with LMODE parameter and RMODE parameter MOID PRIORITY operant{756}, firm RMODE PRIORITY plus one operand{d,e}.

c)* operand : FIRM MODE ADIC operand{d,f}.

d) firm MODE PRIORITY operand{b,d} : firm MODE PRIORITY formula{820e} ; firm MODE PRIORITY plus one operand{d,e}.

e) firm MODE priority NINE plus one operand{b,d} : firm MODE monadic operand{f}.

f) FEAT MODE monadic operand{e,g,h,823b} : FEAT MODE monadic formula{820e,g} ; FEAT MODE secondary{81c}.

g) MOID monadic formula{81b,820d,e,f,g,821a,b,822a,b,c,823a,b,824a,825a,b, c,d,826a,828b} : MOID depression{h,-} ; procedure with RMODE parameter MOID monadic operant{756}, firm RMODE monadic operand{f}.

h) MODE depression{g} : dereference symbol{31c}, soft reference to MODE monadic operand{f}.

{Examples:

b) x + y

d) x × y ; x ;

names. Also f2d3df :=: f5df is not an identity-relation, whereas f2d3df = f5df is a formula, but involves an operation which is not included in the standard-prelude.}

8.4. Formulas

{Formulas are either dyadic, e.g. x + i, or monadic, e.g. abs x. A formula has at least one operator or dereference-symbol. The order of elaboration of a formula is determined by the priority of its operators; monadic formulas are elaborated first and then the dyadic formulas from the highest to the lowest priority. Since the dereference-symbol is not an operator, the programmer is prevented from changing its meaning. }

8.4.1. Syntax

a)* SORTETY formula : SORTETY MOID ADIC formula{b,g,820d,e,f,g}.

b) MOID PRIORITY formula{81b,820d,e,f,g,821a,b,822a,b,c,823a,b,824a,825a, b,c,d,826a,828b} : firm LMODE PRIORITY operand{d}, procedure with LMODE parameter and RMODE parameter MOID PRIORITY operator{43b}, firm RMODE PRIORITY plus one operand{d,e}.

c)* operand : FIRM MODE ADIC operand{d,f}.

d) firm MODE PRIORITY operand{b,d} : firm MODE PRIORITY formula{820e} ; firm MODE PRIORITY plus one operand{d,e}.

e) firm MODE priority NINE plus one operand{b,d} : firm MODE monadic operand{f}.

f) FEAT MODE monadic operand{e,g,h,823b}:FEAT MODE monadic formula{820e,g} ; FEAT MODE secondary{81c}.

g) MOID monadic formula{81b,820d,e,f,g,821a,b,822a,b,c,823a,b,824a,825a,b, c,d,826a,828b} : MOID depression{h,-} ; procedure with RMODE parameter MOID monadic operator{43c}, firm RMODE monadic operand{f}.

h) MODE depression{g} : dereference symbol{31c}, soft reference to MODE monadic operand{f}.

{Examples:

b) x + y

d) x × y ; x ;

8.4.1. continued

f) abs x ;
g) val xx ; abs x ;
h) val xx }

8.4.2. Semantics

a) A formula, other than a depression, is elaborated in the following steps:

Step 1: The formula is replaced by a copy of the routine possessed by the operator-defining occurrence of its operator {7.5.2, 4.3.2.b} ;

Step 2: The copy is treated as a closed-clause and is protected {6.0.2.d} ;

Step 3: The skip-symbol {5.4.2.Step 2} following the equals-symbol following its textually first copied formal-parameter is replaced by a copy of the textually first operand of the formula, and if the operator is not a monadic-operator, then the skip-symbol following the equals-symbol following its textually second copied formal-parameter is replaced by a copy of the textually second operand of the formula ;

Step 4: The elaboration of the copy is initiated; its value, if any, is then that of the formula; if this elaboration is completed or terminated, then the copy is replaced by the formula before the elaboration of a successor is initiated.

b) A depression is elaborated in the following steps:

Step 1: Its operand is elaborated ;

Step 2: If the name obtained in Step 1 is not nil, then the value of the depression is a copy of the value referred to by the name obtained in Step 1 {; otherwise, the further elaboration is undefined}.

The following table summarises the priorities of the operators declared in the standard-priorities (10.2.0).

dyadic									monadic
1	2	3	4	5	6	7	8	9	(10)
minus	∨	∧	=	<	−	×	↑	⊥	¬ − + down lwb
plus			≠	≤	+	÷	lwb		abs bin repr upb
times				≥		÷:	upb		leng short lws
over				>		/	lws		odd sign ups
modb						elem	ups		round entier
prus									re im conj up

8.4.2. continued

Observe that a↑b is not precisely the same as a^b in usual notation; indeed, the value of (-1 ↑ 2 + 4 = 5) and that of (4 - 1 ↑ 2 = 3) both are true, since the first minus symbol is a monadic-operator, whereas the second is dyadic. Although the syntax determines the order in which formulas are elaborated, parentheses may well be used to improve readability; e.g. (a ∧ b) ∨ (¬ a ∧ ¬ b) instead of a ∧ b ∨ ¬ a ∧ ¬ b.

In the formula x + y × 2, both y and 2 are primaries, which allows y to be a firm-priority-SEVEN-operand and 2 to be a firm-priority-EIGHT-operand. The formula y × 2 is then of priority SEVEN. Since x is also a primary, and therefore a firm-priority-SIX-operand, then x + y × 2 is a priority-SIX-formula. The effect of x + y × 2 is thus the same as x + (y × 2).

The operand which follows the dereference-symbol in a depression is soft rather than firm because its elaboration should not involve dereferencing.}

8.5. Cohesions

{Cohesions are of two kinds: nonlocal-generators, e.g. **string**, or selections, e.g. re **of** z. Cohesions are distinct from bases in order that constructions like a **of** b[i] may be parsed without knowing the mode of a and b. Cohesions may not be subscripted or parametrized, but they may be selected from, e.g. father **of** algol in father **of** father **of** algol.}

8.5.0.1. Syntax

a) MODE cohesion{81c,820d,e,f,g,821a,b,822a,b,c,823a,b,824a,825a,b,826a, 828b} : MODE nonlocal generator{851c} ; MODE selection{852a}.

{Examples:

a) **real** (in xx := **real** := 3.14) ; re **of** z }

8.5.1. Generators

{And as imagination bodies forth
The forms of things unknown, the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name.
A Midsummer-night's Dream, William Shakespeare.}

{The elaboration of a generator, e.g. real in xx := real := 3.14 or loc real in ref real x = loc real (usually written real x, by extension 9.2.a) involves the creation of a name, i.e. the reservation of storage. The use of a local-generator implies (with most implementations) the reservation of storage in a run-time stack, whereas nonlocal-generators imply the reservation of storage in another region, term it the "heap", in which garbage-collection techniques may be used for storage retrieval. Since this is usually less efficient, nonlocal-generators should be avoided by the inexperienced programmer. The temptation to use nonlocal-generators unnecessarily, is reduced by the extension 9.2.a, which applies only to local-generators. Local-generators are not cohesions but occur as actual-parameters (see 7.4.1.b) and, therefore, may occur in declarations. }

8.5.1.1. Syntax

a)* generator : MODE LOCAL generator{b,c,-}.
b) reference to MODE local generator{74b,831d} :
    local symbol{31d}, actual MODE declarer{71b}.
c) reference to MODE nonlocal generator{850a} :
    actual MODE declarer{71b}.

{Examples:
b) loc real ;
c) real }

8.5.1.2. Semantics

a) A generator is elaborated in the following steps:
Step 1: Its actual-declarer is elaborated {7.1.2.c} ;
Step 2: The value of the generator is the value {name} obtained in Step 1.

b) The scope {2.2.4.2} of the value of a local-generator is the smallest range containing that generator; that of a nonlocal-generator is the program.

{The closed-clause
(ref real xx ; (ref real x = real := pi ; xx := x) ; xx = pi)
possesses the value true, but the closed-clause
(ref real xx ; (real x := pi ; xx := x) ; xx = pi}
possesses an undefined value since the name referred to by the name possessed by xx becomes undefined upon the completion of the elaboration of the inner

8.5.1.2. continued

range, which is the scope of the name possessed by x (7.0.2). The closed-clause

((ref real xx ; real x := pi ; xx := x) = pi)

however, possesses the value true. }

8.5.2. Selections

{A selection selects a field from a structured value; e.g., re of z selects the first real field (usually termed the real part) of the value possessed by z. If z is a name, then re of z is also a name, but if w is a complex value, then re of w is a real value, not the name referring to a real value. }

8.5.2.1. Syntax

a) REFETY MODE selection{850a} : MODE named TAG selector{71j}, of symbol{31e}, weak REFETY structured with LFIELDSETY MODE named TAG RFIELDSETY secondary{81c}.

{Examples: The following examples are assumed in the reach of the declarations:

```
 struct language = (int age, ref language father) ;
 language algol := (10, language := (14, nil)) ;
 language pl1 = (4, algol) ;
a)  age of pl1 ; father of algol }
```

{Rule a ensures that the value of the secondary has a field selected by the field-selector in the selection (see 7.1.1.e,f,h,j and the remarks below 7.1.1 and 8.5.2.2). ~~The occurrence of~~ An identifier which is the same sequence of symbols as a field-selector in the same reach creates no ambiguity. Thus age of algol := age is a (possibly confusing to the human) assignation if the second occurrence of age is an integral-mode-identifier.}

8.5.2.2. Semantics

A selection is elaborated in the following steps:

Step 1: Its secondary is elaborated, and the structured value which is, or is referred to by, the value of that secondary is considered ;

Step 2: If the value of the secondary is a name, then the value of the selection is a new instance of the name which refers to that field of the considered structured value selected by its field-selector; otherwise, it is a new instance of {the value which is} that field itself.

{In the examples of 8.5.2.1, age of algol is a reference-to-integral-selection, and, by 8.5.0.1.a, a reference-to-integral-cohesion, but age of pl1 is an integral-selection and an integral-cohesion. It follows that age of algol may appear as a destination (8.3.1.1.e) in an assignation but age of pl1 may not. Similarly, algol is a reference-to-[language]-base but pl1 is a [language]-base and no assignment may be made to pl1. (Here [language] stands for structured-with-integral-field-[age]-and-reference-to-[language]-field-[father] and [age] stands for letter-a-letter-g-letter-e etc.) The selection father of pl1, however, is a reference-to-[language]-selection and thus a reference-to-[language]- cohesion whose value is the name possessed by algol. It follows that the identity-relation father of pl1 :=: algol possesses the value true. If father of pl1 is used as a destination in an assignation, there is no change in the name which is a field of the structured value possessed by pl1, but there may well be a change in the [language] referred to by that name. By similar reasoning and because the operators re and im possess routines (10.2.5.b,c) which deliver values whose mode is 'real' and not 'reference to real', re of z := im w is an assignation, but re z := im w is not. }

8.6. Bases

{Bases are denotations, e.g. 3.14, identifiers, e.g. x, slices, e.g. x1[i] and calls, e.g. sin(x). Bases are generally elaborated first. They may be subscripted, parametrized and selected from and are often used as operands.}

8.6.0.1. Syntax

a) MOID base{81d,820d,e,f,g,821a,b,822a,b,c,823a,b,824a,825a,b,c,d,826a, 828b} : MOID slice{861a,-} ; MOID call{862a} ; MOID denotation{510b, 511a,512a,513a,514a,52a,26,53a,54b,55a,-} ; MOID identifier{41b,-}.

8.6.1.1. continued

i) subscript{b,c,e} : strong integral unit{61e}.
j)* trimscript : trimmer{f} option ; subscript{i}.
k)* indexer : ROWS leaving ROWSETY indexer{b,c,d,e}.

{Examples:
a) x1[i] ; x2[i,j] ; x2[i] ; x1[2:n] ;
b) 2:n,j ; 1,2:n ;
c) i,j ;
d) 2:n ;
e) i ;
f) 2:n ; 2:n at 0 ;
g) at 0 ;
h) 0 ;
i) i }

{In rule a, 'ROWS' reflects the number of trimscripts in the slice, 'ROWSETY' the number of these which are trimmer-options and 'ROWWSETY' the number of 'row of' not involved in the indexer. In the slices x2[i,j], x2[i,2:n], x2[i], these numbers are (2,0,0), (2,1,0) and (1,0,1) respectively. Because of rules d and 7.1.1.t, 2:3at0 ; 2:n ; 2: ; :5 and :at0 are trimmers, while rules b and d allow trimmers to be omitted. }

8.6.1.2. Semantics

A slice is elaborated in the following steps:

Step 1: Its primary, and all constituent strict-lower-bounds, strict-upper-bounds and new-lower-bounds of its indexer are elaborated collaterally {6.2.2.a} ;

Step 2: The multiple value which is, or is referred to by, the value of the primary, is considered, a copy is made of its descriptor, and all the states {2.2.3.3.b} in the copy are set to 1 ;

Step 3: The trimscript following the sub-symbol is considered, and a pointer, "i", is set to 1 ;

Step 4: If the considered trimscript is not a subscript, then Step 5 is taken; otherwise, letting "k" stand for its value, if $l_i \leq k \leq u_i$, then the offset in the copy is increased by $(k - l_i) \times d_i$, the i-th quintuple is "marked", and Step 6 is taken; otherwise, the further elaboration is undefined ;

8.6.1.2. continued

Step 5: The values "l", "u" and "l'" are determined from the considered trimscript as follows:

if the considered trimscript contains a strict-lower-bound (strict-upper-bound), then l (u) is its value; otherwise, l (u) is l_i (u_i); if it contains a new-lower-bound, then l' is its value; otherwise, l' is 1 ;

if now $l_i \leq l$ and $u \leq u_i$, then the offset in the copy is increased by $(l - l_i) \times d_i$, and then l_i is replaced by l' and u_i by $(l' - l) + u$; otherwise, the further elaboration is undefined ;

Step 6: If the considered trimscript is followed by a comma-symbol, then the trimscript following that comma-symbol is considered instead, i is increased by 1, and Step 4 is taken; otherwise, all quintuples in the copy which were marked by Step 4 are removed, and Step 7 is taken ;

Step 7: If the copy now contains at least one quintuple, then the multiple value composed of the copy and those elements of the considered value which it describes and whose mode is that obtained by deleting 'slice' and the initial 'reference to', if any, from that notion ending with 'slice' of which the slice is a terminal production, is considered instead; otherwise, the element of the considered value selected by {the index equal to} the offset in the copy is considered instead ;

Step 8: If the value of the primary is a name, then the value of the slice is a new instance of the name which refers to the considered value, and, otherwise, is a new instance of the considered value itself.

{A trimmer restricts the possible values of a subscript and changes its notation: first, the value of the subscript is restricted to run from the value of the strict-lower-bound to the value of the strict-upper-bound, both given in the old notation; next, all restricted values of that subscript are changed by adding the same amount to each of them, such that the lowest value then equals the value of the new-lower-bound. Thus, the assignations y1[1:n-1] := x1[2:n] ; y1[n] := x1[1] ; x1 := y1 effect a cyclic permutation of the elements of x1. }

8.6.2. Calls

8.6.2.1. Syntax

a) MOID call{860a} : firm procedure with PARAMETERS MOID primary{811d}, actual PARAMETERS{54e,74b} pack.

8.6.2.1. continued

{Examples:
a) sin(x) }

8.6.2.2. Semantics

A call is elaborated in the following steps:

Step 1: Its primary is elaborated and a copy is made of {the routine which is} its value ;

Step 2: The call is replaced by that copy ;

Step 3: That copy is treated as a closed-clause and is protected {6.0.2.d} ;

Step 4: The copy as possibly modified by Step 3 is further modified by replacing the skip-symbols following the equals-symbols following the copied formal-parameters {5.4.2.Step 2} in the textual order by the actual-parameters of the call taken in the same order ;

Step 5: The elaboration of the copy is initiated; its value, if any, is that of the call; if this elaboration is completed or terminated, then the copy is replaced by the call before the elaboration of a successor is initiated.

The call samelson(m, (int j) real : x1[j]) as contained in the reach of the declaration

```
proc samelson = (int n, proc(int)real f)real :
  begin long real s := long 0 ; for i to n do s plus leng f(i) ↑ 2 ;
  short long sqrt(s) end
```

is elaborated by considering (Step 1) the closed-clause

```
(val(int n = skip, proc(int)real f = skip ; real :=
  begin long real s := long 0 ; for i to n do s plus leng f(i) ↑ 2 ;
  short long sqrt(s) end)).
```

Supposing that n, s, f and i do not occur elsewhere in the program, this closed-clause is protected (Step 3) without further alteration. The actual-parameters are now inserted (Step 4), yielding the closed-clause

```
(val(int n = m, proc(int)real f = (int j)real : x1[j] ; real :=
  begin long real s := long 0 ; for i to n do s plus leng f(i) ↑ 2 ;
  short long sqrt(s) end)) ,
```

and this closed-clause is elaborated (Step 5). Note that, for the duration of this elaboration, n possesses the same integer as that referred to by the name possessed by m, and f possesses the same routine as that possessed by the routine-denotation ((int j)real : x1[j]). During the elaboration of this and its inner nested closed-clauses (9.3), the elaboration of f(i) itself involves the elaboration of the closed-clause (val(int j = i ; real := x1[j])), and, within this inner closed-clause, the first occurrence of j possesses the same integer as that referred to by the name possessed by i. }

9. Extensions

a) An extension is the insertion of a comment between two symbols or the replacement of a certain sequence of symbols, possibly satisfying certain restrictions, by another sequence of symbols.

b) No extension may be performed within a comment {3.0.9.b} or row-of-character-denotation {5.3}.

c) Some extensions are given in the representation language, except that
A, B and C stand for strong-integral-unitary-clauses {8.1.1.a},
D for a strong-unitary-boolean-clause {8.1.1.a},
E for a strong-unitary-void-clause {8.1.1.a},
F and G for unitary-clauses {8.1.1.a},
H for a unitary-clause-list-proper {8.1.1.a},
I, J, K, L for mode-identifiers {4.1.1.b},
M and M1 for label-identifiers {4.1.1.b},
N for a mode-identifier-option {4.1.1.b},
O for a conformity-relator {8.3.2.1},
P for an indication {4.2.1.a},
Q for a virtual-plan {5.4.1.c,d},
R for a routine-denotation {5.4.1.a},
S for the standard-prelude {2.1.b,10} if the extension is performed outside the standard-prelude and, otherwise, for the empty sequence of symbols,
T for a condition followed by a choice-clause {6.4.1.c,d},
U for a declarer {7.1.1.a},
V for a formal-declarer {7.1.1.b} all of whose formal-row-of-rowers {7.1.1.q} are empty,
W for a tertiary {8.1.1.b},
X and Y for a soft-reference-to-tertiary {8.1.1.b}, and
Z for a soft-reference-to-tertiary-list-proper {8.1.1.b}.

d) Each representation of a symbol appearing in sections 9.1 up to 9.5 may be replaced by any other representation, if any, of the same symbol.

9.1. Comments

{A source of innocent merriment.
Mikado, W.S. Gilbert. }

A comment {3.0.9.b} may be inserted between any two symbols {but see 9.b}.

{e.g., (m > n | m | n) may be replaced by
(m > n | m **c** the larger of the two **c** | n).}

9.2. Contractions

a) **ref** VI = **loc** U where U and V specify the same mode {7.1.2.a}may be replaced by UI.

{e.g., **ref** **real** x = **loc** **real** may be replaced by **real** x and **ref** **bool** p = **loc** **bool** := **true** may be replaced by **bool** p := **true**.}

b) **mode** P = **struct** may be replaced by **struct** P = and **mode** P = **union** by **union** P =.

{e.g., **mode** **compl** = **struct**(**real** re, im) (see also 9.2.c) may be replaced by **struct** **compl** = (**real** re, im).}

c) If a given mode-declaration {7.2.1.a} (priority-declaration {7.3.1.a}, identity-declaration {7.4.1.a}, operation-declaration {7.5.1.a}, formal-parameter {5.4.1.h}, field-declarator {7.1.1.g} and another one following a comma-symbol {3.1.1.e} following the given one both begin with a mode-symbol, structure-symbol, union-of-symbol, priority-symbol, operation-symbol {all 3.1.1.d}, one same actual-declarer or formal-declarer {both 7.1.1.b}, then the second of these occurrences may be omitted.

{e.g., **real** x, **real** y := 1.2 may be replaced by **real** x, y := 1.2, but **real** x, **real** y = 1.2 may not be replaced by **real** x, y = 1.2, since the first occurrence of **real** is an actual-declarer whereas the second is a formal-declarer. Note also that **mode** **b** = **bool**, **mode** **r** = **real** may be replaced by **mode** **b** = **bool**, **r** = **real**, etc.}

d) If an actual-parameter {7.4.1.b} (source {8.3.1.1.f}) is a routine-denotation {5.4.1.a} (routine-denotation not beginning with (:), then its first open-symbol and last close-symbol {both 3.1.1.e} may simultaneously be omitted.

{e.g., op + = ((int a)int : a) may be replaced by op + = (int a) int : a.}

e) If each corresponding pair of constituent-declarers in Q and R specifies the same mode, then proc QI = R may be replaced by proc I = R, op QP = R by op P = R, and proc QN := R by proc N := R.

{e.g., proc (ref int) incr = (ref int i): i plus 1 may be replaced by proc incr = (ref int i) : i plus 1, op (ref int) int decr = (ref int i) int : i minus 1 may be replaced by op decr = (ref int i) int : i minus 1, and proc (real) int p = (real x) int : round x, obtained by 9.2.a,d from ref proc (real) int p = loc proc (real) int := ((real x) int : round x) may be replaced by proc p := (real x) int : round x.}

f) [:] may be replaced by [] , [:, by [, , ,:, by ,, , ,:] by ,] , [:at by [at, and ,:at by ,at.

{e.g., [:] real may be replaced by [] real.}

9.3. Repetitive Statements

a) The unitary-statement {6.0.1.c}

```
begin int J := A, int K = B, L = C;
  M : if S (K>0|J≤L|:K<0|J≥L|true)
      then int I = J; (D|E; (SJ:= J + K); go to M)
  end,
```

where J, K, L and M do not occur in D, E or S, and where I differs from J and K, may be replaced by

```
  for I from A by B to C while D do E,
```

and if, moreover, I does not occur in D or E, then for I from may be replaced by from.

9.3. continued

b) The unitary-statement {6.0.1.c}

```
    begin int J :≔ A , int K = B;
      M : (int I = J; (D|E; (S.J := J + K); go to M))
    end,
```

where J, K and M do not occur in D, E or S, and where I differs from J and K, may be replaced by

  for I **from** A **by** B **while** D **do** E,

and if, moreover, I does not occur in D or E, then **for** I **from** may be replaced by **from**.

c) **from** 1 **by** may be replaced by **by**, **by** 1 **to** by **to**, **by** 1 **while** by **while**, and **while** **true** **do** by **do**.

{e.g., **for** i **from** 1 **by** 1 **to** n **while** **true** **do** x **plus** a may be replaced by **to** n **do** x **plus** a. Note that **to** 0 **do** S and **while** **false** **do** S do not cause S to be elaborated at all, whereas **do** S causes S to be elaborated repeatedly until it is terminated or interrupted.}

9.4. Contracted conditional clauses

{The flowers that bloom in the spring, Tra la,
Have nothing to do with the case.
Mikado, W.S. Gilbert. }

a) **else** **if** T **fi** **fi** may be replaced by **elsf** T **fi** and
then **if** T **fi** **fi** by **thef** T **fi**.

{e.g., **if** p **then** princeton **else** **if** q **then** grenoble **else** zandvoort **fi** **fi** may be replaced by **if** p **then** princeton **elsf** q **then** grenoble **else** zandvoort **fi** or by (p | princeton |: q | grenoble | zandvoort). Many more examples are given in 10.5.}

b) (**int** I = A ; **if** SI = 1 **then** F **elsf** S(I = 2 | **true**) **then** G **fi**), where I does not occur in F,G or S, may be replaced by **case** A **in** F, G **esac** {or by (A| F, G)}.

c) (**int** I = A ; **if** SI = 1 **then** F **else** **case** (SI = 1) **in** H **esac** **fi**), where I does not occur in F, H or S, may be replaced by **case** A **in** F, H **esac** {or by (A | F, H)}.

~~{Examples of the use of such "case" clauses are given in 11.11.w, ap.}~~

9.4. continued

d) (((X O W | M), (Y O W | M 1)) ; ((<u>false</u> | <u>true</u>)| <u>skip</u>). M : F .M1 : G), where M and M1 do not occur in F and/or G, may be replaced by
<u>case</u> X, Y O W <u>in</u> F, G <u>esac</u> {or by (X, Y O W | F, G)}.

e) (((X O W | M) , (X, Z O W | H)). M : F), where M does not occur in F and/or H, may be replaced by
<u>case</u> X, Z O W <u>in</u> F, H <u>esac</u> {or by (X, Z O W | F, H)}.
{Examples of the use of such "case" clauses are given in 11.11.w,ap.}