STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

REKENAFDELING

WORKING DOCUMENT ON THE ALGORITHMIC LANGUAGE ALGOL 68

A. van Wijngaarden

B.J. Mailloux

J.E.L. Peck

C.H.A. Koster

MR 95
juli 1968

1. Language and metalanguage {25 July 1968}

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", "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 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.

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
'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.Step 3, 4).}

1.1.3. The syntax of the metalanguage {10 July 1968}

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 sequence of "metamembers" separated by blanks; a metamember is either a metanotion and is then said to be productive, or is a nonempty sequence of small syntactic marks.

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 1.2.1.1, '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.m,n). In the production rule 'EMPTY : .' (1.2.1.i), the metanotion 'EMPTY' has a direct production which is an empty list of metanotions. }

1.1.4. The production rules of the metalanguage {10 July 1968}

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.n}, each of whose direct productions is another small letter, may be added.

{For instance, the rule
'TAG : LETTER ; TAG LETTER ; TAG DIGIT.',
from 1.2.1.1 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 if 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 {10 July 1968}

The production rules of the strict language are 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 each 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 spaces and hyphens in the given rule are removed and the rule so obtained is a production rule of the strict language ;

Step 3: A number of production rules may be added for the notions 'other mode indication' and 'other operator indication' {4.2.1.b,e,f} each of whose direct productions is a symbol different from an other symbol ;

Step 4: 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.b} 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 option.'
derived from 7.1.1.r by Step 1 is used in Step 2 to provide two production rules of the strict language, viz.
'actuallowerbound:strictlowerboundoption.' and
'actualupperbound:strictupperboundoption.' ;
however, to ease the burden on the reader, who may more easily ignore spaces himself, some spaces 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 option.' and
'actual upper bound : strict upper bound option.'.

Note that
  'actual lower bound : strict upper bound option.'
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 {25 July 1968}

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 consisting of a second sequence of symbols preceded and/or followed by (a) nonemtpy sequence(s) of symbols "contains" that second sequence of symbols.

c) A "paranotion" at an occurrence not 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

iii) a notion whose prodution 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" (8.6.1.1.1) 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 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" (8.2.6.1.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), 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 begining with an asterisk have been included in order to shorten the semantics. }

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

e) A protonotion which is a member of a (direct) production of a notion is a "(direct) consituent" of that notion, provided that it is not also a constituent of either that notion or that protonotion {, e.g., 'digit zero' is a direct constituent of 'integral denotation' (5.1.1.a) and 'digit one' is a constituent of 'integral denotation' but not a direct constituent (5.1.1.b). For examples involving the proviso, see the remarks following section f}.

f) A paranotion which denotes protonotions all of which are (direct) constituents of notions denoted by a second paranotion is a (direct) constituent of that second paranotion. {e.g., since paranotions stand for 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) If something is left undefined or is said to be undefined, this means that it is not determined by this Report alone, and that, for its determination, information from outside this Report has to be taken into account.

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 production 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 elaboration (value, mode, scope) a reference-to-real-nonlocal-assignation.}

1.1.7. The extended language {25 July 1968}

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., real x, y, z means the same as real x, real y, real 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 representation 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. , begin, **begin**, and 'BEGIN' are all representations of the begin-symbol in the representation language.}

1.2. The metaproduction rules {10 July 1968}

1.2.1. Metaproduction rules of modes

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

{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 {25 July 1968}

a) PRIMITIVE : integral ; real ; boolean ; character ; format.
b) ROWS : row of ; row of ROWS.
c) ROWSETY : ROWS ; EMPTY.
d) ROWWSETY : ROWSETY.
e) NONROW : TYPE ; reference to MODE ; UNITED.
f) REFETY : reference to ; EMPTY.
g) NONREF : TYPE ; row of MODE ; UNITED.
h) NONPROC : PLAIN ; structured with FIELDS ;
   procedure with PARAMETERS MOID ; format ;
   row of NONPROC ; UNITED ; reference to NONPROC.
i) LMODE : MODE.
j) RMODE : MODE.
k) LMODESETY : MODES and ; EMPTY.
l) RMODESETY : and MODES ; EMPTY.
m) LFIELDSETY : FIELDS and ; EMPTY.
n) RFIELDSETY : and FIELD ; EMPTY.
o) COMPLEX : structured with real named letter r letter e
   and real named letter i letter m.
p) STRING : row of character ; character.
q) MABEL : 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) STRONG : strong ; coFIRM.
i) FIRM : firm ; weak ; soft.
j) STIRM : strong firm.
k) ADAPTED : ADJUSTED ; widened ; arrayed ; hipped ; voided.
l) ADJUSTED : FITTED ; procedured ; united.
m) FITTED : dereferenced ; deprocedured.

1.2.4. Metaproduction rules associated with formulas {25 July 1968}

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) ANY : sign ; zero ; digit ; point ; exponent ; complex ;
character ; suppressible ANY ; replicatable ANY.
e) NOTION : ALPHA ; NOTION ALPHA.
f) SEPARATORETY : comma symbol ; go on symbol ; completer ;
sequencer ; statement interlude option ; EMPTY.

{Rule e 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,e,f.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 otherwise 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 cross-referencing system which appears under Syntax uses the following conventions in order to save space:
i) all points are omitted, e.g. "3.0.6.a" appears as "306a",
ii) redundant 1's are omitted, e.g. "811a" appears as "81a",
iii) some dead-ends are marked by "-".

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 ; bits t ; string s ; 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 ;
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 = void : (l:l) ;
exit: princeton : grenoble : st pierre de chartreuse ;
kootwijk : warsaw : zandvoort : amsterdam : tirrenia :
north berwick : x := 1.}
```

2. The computer and the program {22 July 1968}

{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{30i}, strong serial void clause{61a}, close{30j}.

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.g} 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. 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 of {1.1.6.f} and "to identify".

c) A given occurrence of an "identifier" {4.1} ("indication" {4.2}, "operator" {4.3}) may identify a "defining" ("indication-defining", "operator-defining") occurrence of the same identifier (indication, operator).

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

e) An external object considered as a terminal production of a given notion may possess a value, called "the" value of the external object when it is clear which notion is intended.

f) An 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} ("operation-declaration" {7.5}) 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. a clause (6.1.1)} 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".

i) A value may be of the same mode as another value; this relationship is permanent.

j) A value may be equivalent to another value {2.2.3.1.d,f} and a value may be smaller than another value {10.2.2.a, 10.2.3.a}. 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 given value is a component of another value 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 value 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 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

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

2.2.3.1. Plain values {22 July 1968}

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 lengths of integers (real numbers) {10.1.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 {10.2.2.a, 10.2.3.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 {10.2.2.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 {10.2.3.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 integral or real number. The inverse transformations are only possible on those real numbers which are equivalent to a value of smaller length number.

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

f) Each character has an "integral equivalent" {10.1.h}. i.e. a nonnegative 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, $i = 1, \ldots, n$, $(u_i - l_i) \times d_i < 0$, then the number of elements in the multiple value is zero; otherwise, it is

$$(|u_1 - l_1| + 1) \times d_1 + \ldots + (|u_n - l_n| + 1).$$

The descriptor "describes" each element for which there exists an n-tuple $(r_1, \ldots, r_n)$ of integers satisfying, for each $i = 1, \ldots, n$, $l_i \leq r_i \leq u_i$, if $d_i > 0$, or $u_i \leq r_i \leq l_i$, if $d_i < 0$, and that ~~the~~ 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 referred to by the value of a slice {8.6.1} the value of whose primary {8.6.1.1.a} 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}).

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.Step8, and refers to precisely one instance of a value}.

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) Each instance {2.2.1} of a value is of one specific mode which is a terminal production of 'NONUNITED' {1.2.1.b}; furthermore, all instances of a given value other than nil {2.2.3.5.a} are of one same mode.

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').

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 'named' followed by a terminal production of 'TAG' {1.2.1.l} whose terminal production {field-selector} selects {2.2.3.2} that field.

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, 5.5} is the smallest range {4.1.1.e} containing a defining {4.1.2.a} (indication-defining {4.2.2.a}, operator-def-

ining {4.3.2.a} occurrence of an identifier (indication, operator), if any, applied but not defined (indication-applied but not indication-defined, operator-applied but not operator-defined) within 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", "serial" 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.

{What actions, if any, are elementary is left undefined, except as provided in 6.4.2.b.}

2.3. Semantics

{"I can explain all the poems that ever were invented, - and a good may 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 sequence of symbols are programs and the Semantics which actions are performed by the computer when elaborating a program. Both Syntax and Semantics are recursive.}

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 catagory 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 called 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 a certain number of deletions or insertions of symbols and not by a smaller number could be regarded as a program with that number of syntactical errors. Any program that can be obtained by performing that number of deletions or insertions may be called a "posibly 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 of 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 occurrences of 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.f) (mode-declarations (7.2) or priority-declarations (7.3), captions (7.5.1.b)) containing those defining (indication-defining, operator-defining) occurrences.}

3. Basic tokens and general constructions {23 July 1968}

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 SEPARATORETYs{c,d,f} :
  NOTION ; NOTION, SEPARATORETY{31b,f,61d,l},
      chain of NOTIONs separated by SEPARATORETYs{c}.
d) NOTION list{e} :
  chain of{c} NOTIONs separated by comma symbols{31e}.
e) NOTION array :
  NOTION, comma symbol{31e}, NOTION list{d}.
f) NOTION sequence :
  chain of{c} NOTIONs separated by EMPTYs.
g) NOTION pack :
  open symbol{31e}, NOTION, close symbol{31e}.
h) NOTION box :
  open{i}, NOTION, close{j} ;
  open{i}, NOTION, close{j}, TAG{302b,41c,d}.
i) open{h,2d} : open symbol{31e} ; begin symbol{31e}.
j) close{h,2d} : close symbol{31e} ; end symbol{31e}.

{Examples:
a) a ; 0 ; + ; int ; if ; . ; nil ; for ; " ;
b) 0 ; c) 0, 1, 2 ;
d) 0 ; 0, 1, 2 ; e) 0, 1 ;
f) 0 ; 000 ; g) (1, 2, 3) ;
h) (x := 1 ; y := 2) ; (stop)that is the end ;
  begin x := 1 ; y := 2 end assignations ;
i) (; begin ; j)) ; end }

3.0.2. Letter tokens

{25 July 1968}

a) letter token : LETTER {b}.
b) LETTER{a,30h,309d,41b,c,d} : 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) and row-of-character-denotations (5.3). }

3.0.3. Denotation tokens

a) denotation token : number token{b} ; true symbol{31b} ;
   false symbol{31b} ; formatter symbol{31b} ;
   routine symbol{31b} ; void 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,512d} : DIGIT symbol{31b}.
e) flipflop{52a} : flip symbol{31b} ; flop symbol{31b}.

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

{Denotation-tokens are constituents of denotations (Chapter 5). Some denotation-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 {25 July 1968}

a) action token : operator token{b} ; equals symbol{31c} ;
value of symbol{31c} ; confrontation token{d}.

b) operator token{42e} : or symbol{31c} ; and symbol{31c} ;
not 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} ; over symbol{31c} ;
quotient symbol{31c} ; modulo symbol{31c} ;
absolute value of symbol{31c} ; lengthen symbol{31c} ;
shorten symbol{31c} ; round symbol{31c} ; sign symbol{31c};
entier symbol{31c} ; odd symbol{31c} ;
representation symbol{31c} ; real part of symbol{31c} ;
imaginary part of symbol{31c} ; conjugate symbol{31c} ;
binal symbol{31c} ; to the power symbol{31c} ;
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} ;
prus and becomes symbol{31c} ;
up symbol{31c} ; down symbol{31c}.

c) plusminus{512h} : 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) ∨ ; ∧ ; ¬ ; ≠ ; < ; ≤ ; ≥ ; > ; + ; × ; / ; ÷ ; ÷: ; abs ;
leng ; short ; round ; sign ; entier ; odd ; repr ; re ;
im ; conj ; bin ; ↑ ; minus ; plus ; times ; over ;
modb ; prus ; up ; down ;

c) + ; - ;

d) := ; :: ; ::= ; :=: ; :≠: }

{Operator-tokens are constituents of formulas (8.4.1). An operator-token may be caused to possess an operation by the elaboration of an operation-declaration (7.5). Confrontation-tokens are constituents of confrontations (8.3).}

3.0.5. Declaration tokens

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

{Examples:
a) **int** ; **long** ; **ref** ; **proc** ; **struct** ; **union** ; **loc** ; **compl** ;
bits ; **string** ; **file** ; **mode** ; **priority** ; **op** }

{Declaration-tokens either are or are constituents of declarers (7.1), which specify modes (2.2.4), or of declarations (7.2, 3, 4, 5).}

3.0.6. Syntactic tokens

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

{Examples:
a) (; **begin** ;) ; **end** ; , ; **elem** ; **par** ; [;] ; : ; **at** ;
by ; **if** ; **then** ; **fi** ; **of** ; : }

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

3.0.7. Sequencing tokens {25 July 1968}

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 (6.1.2).}

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} ;
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} ;
plus i times symbol{31h}.

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

c) comment item : character token{d} ;
other comment item{1.1.5.Step 4}.

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

{Examples:
a) 'for' ; from ; to ; while ; do ; thef ; elsf ; case ; in ;
esac ; i ;

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} ;
other mode indication{1.1.5.Step 3} ;
other operator indication{1.1.5.Step 3}.

{Examples:
a) " ; **c** ; **primitive** ; ? }

3.1. Symbols {26 July 1968}

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

b) Denotation tokens

symbol	representation
digit zero symbol{303d}	0
digit one symbol{303d,73b}	1
digit two symbol{303b,73c}	2
digit three symbol{303b,73d}	3
digit four symbol{303b,73e}	4
digit five symbol{303b,73f}	5
digit six symbol{303b,73g}	6
digit seven symbol{303b,73h}	7
digit eight symbol{303b,73i}	8
digit nine symbol{303b,73j}	9
point symbol{303b,512b,d}	.
times ten to the power symbol{303b,512g}	$_{10}$ e

symbol	representation	
true symbol{513a,71w}	true	
false symbol{513a,71w}	false	
formatter symbol{55a}	f	
routine symbol{54b}	:	expr
void symbol{54g}	void	
flip symbol{303e}	1	
flop symbol{303e}	0	
space symbol{309d}	.	

c) Action tokens

symbol	representation	
or symbol{304b}	∨	or
and symbol{304b}	∧	and
not symbol{304b}	¬	not
equals symbol{42e,72a,73a,74a,75a}	=	eq
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
plus symbol{304c}	+	
minus symbol{304c}	-	
times symbol{304b}	×	*
over symbol{304b}	/	
quotient symbol{304b}	÷	quotient
modulo symbol{304b}	÷:	mod
absolute value of symbol{304b}	abs	
lengthen symbol{304b}	leng	
shorten symbol{304b}	short	
round symbol{304b}	round	
sign symbol{304b}	sign	
entier symbol{304b}	entier	
odd symbol{304b}	odd	

symbol	representation		
representation symbol{304b}	repr		
real part of symbol{304b}	re		
imaginary part of symbol{304b}	im		
conjugate symbol{304b}	conj		
binal symbol{304b}	bin		
to the power symbol{304b}	↑	power	*:
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		
up symbol{304b}	up		
down symbol{304b}	down		
value of symbol{84h}	val		
becomes symbol{831b}	:=	←	
conforms to symbol{832b}	::	ct	
conforms to and becomes symbol{832b}	::=	ctb	
is symbol{833b}	:=:	is	
is not symbol{833b}	:≠:	is not	isnot

d) Declaration tokens

symbol	representation
integral symbol{71c,v}	int
real symbol{71c}	real
boolean symbol{71c,w}	bool
character symbol{71c}	char
format symbol{71c}	format
long symbol{42c,e,f,510b,52a,71d}	long
reference to symbol{71l,m,n}	ref
procedure symbol{71x}	proc
structure symbol{71e,k}	struct

symbol	representation
union of symbol{71aa}	union
local symbol{851b}	loc
complex symbol{42c}	compl
bits symbol{42c}	bits
string symbol{42c}	string
file symbol{42c}	file
mode symbol{72a}	mode
priority symbol{73a}	priority
operation symbol{75b}	op

e) Syntactic tokens

symbol	representation	
open symbol{21a,30g,i,5513a,309d,	(	
begin symbol{30i}	begin	
close symbol{21a,30g,j,5513a,309d,	)	
end symbol{30j}	end	
comma symbol{30b,d,e,5513a,54e,62e,g, 309d,71f,p,ab,861b,c}	,	comma
elementary symbol{63a}	elem	
parallel symbol{62b,c,d,f}	par	
sub symbol{71o,861a}	[	(
bus symbol{71o,861a}	]	)
up to symbol{71q,861a}	:	
at symbol{861g}	at	
by symbol{861i}	by	
if symbol{64a}	(	if
then symbol{64e}	|	then
else symbol{64e}	|	else
fi symbol{64a}	)	fi
of symbol{852a}	of	
label symbol{61k}	:	

f) Sequencing token

Sequencing token	representation	
go on symbol{30c,61c,d,j}	;	
completion symbol{61l}	.	**exit**
go to symbol{61d,82d}	**go to**	**goto**

g) Hip tokens

symbol	representation
skip symbol{82c}	**skip**
nil symbol{827e}	**nil**

h) Extra tokens

symbol	representation	
for symbol	**for**	
from symbol	**from**	
to symbol	**to**	
while symbol	**while**	
do symbol	**do**	
then if symbol	|:	**thef**
else if symbol	|:	**elsf**
case symbol	(	**case**
in symbol	|	**in**
esac symbol	)	**esac**
plus i times symbol{309d}	⊥	**i**

i) Special tokens

symbol	representation	
quote symbol{514a,53a,b,d}	"	
comment symbol	<u>c</u>	<u>comment</u>

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.}
```

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 letter-tokens {1.1.4.Step 2}, other-mode-indications and other-operation-indications {1.1.5.Step 3}, other-comment-items and other-string-items {1.1.5.Step 4} may be added, provided that no letter-token or indication {4.2} has the same representation as any other basic-token {3.0.1.a}, and that no comment-item {3.0.9.c} (string-item {5.3.1.b}) has the same representation as any other comment-item or the comment-symbol (any other string-item or the quote-symbol).

d) The fact that representations of the letter-tokens 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 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 {24 July 1968}

{A proper program is a program satisfying the context conditions, e.g. if (<u>real</u> x ; x := 1) is contained in a proper program, then the second occurrence of x is a reference-to-real-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 <u>mode</u> <u>a</u> = <u>a</u>.}

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-identifiers, are made to possess values by the elaboration of identity-declarations (7.4). Some identifiers possessing values which are not names might, in other languages, be called constants, e.g. m in <u>int</u> 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-identifiers, possess values which are or are not names.}

4.1.1. Syntax

a)* identifier : MABEL identifier{b}.
b) MABEL identifier{54f,61k,71v,w,827d,860a} : TAG{c,d,302b}.
c) TAG LETTER{b,c,d,30h,71h} : TAG{c,d,302b}, LETTER{302b}.
d) TAG DIGIT{b,c,d,30h,71h} : 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.r 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 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 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 identifiers and then to discover which defining occurrence is identified by a given applied occurrence. }

a) A given occurrence of an identifier defines if
i) it follows a formal-declarer {5.4.1.f}, or
ii) within some range, it is the textually first occurrence of that identifier in a constituent lower-bound-interrogation or upper-bound-interrogation or lower-state-interrogation or upper-state-interrogation {7.1.1.v,w} of that range, or
iii) it is contained in a label {6.1.1.k} ;
otherwise, it is "applied".

b) If a given occurrence of an identifier is applied, then it may identify a defining occurrence found by the following steps:
Step 1: The given occurrence is called 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 called the home and Step 3 is taken {; other-

wise, there is no defining occurrence which the given occurrence identifies} ;

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

{In the closed-clause (bits x := 101 ; abs x[bits width] = 1), the first occurrence of x is a defining occurrence of a reference-to-row-of-boolean-identifier. The second occurrence of x identifies the first and, in order to satisfy the identification condition (4.1.1), is also 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.Step 3, 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,72a} ;
      other mode indication{1.1.5.Step 3}.
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}.
d)* priority indication : PRIORITY indication{e}.
e)  PRIORITY indication{43c,73a} :
      long symbol{31d} sequence option, operator token{304b} ;
      long symbol{31d} sequence option, equals symbol{31c} ;
      other operator indication{1.1.5.Step 3}.
```

f) monadic indication{43g} :
   long symbol{31d} sequence option, operator token{304b} ;
   other operator indication{1.1.5.Step 3}.
g)* adic indication : ADIC indication{e,f}.

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

4.2.2. Identification of indications

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

a) A given occurrence of an indication indication-defines if it precedes the equals-symbol of a mode-declaration {7.2} or priority-declaration {7.3}; otherwise, it is "indication-applied".

b) If a given occurrence of an indication is indication-applied, then it may identify an indication-defining occurrence of the indication found 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 indication, then the given occurrence identifies it; otherwise, Step 2 is taken.".

{Indications have no inherent meaning. The indication-defining occurrence of an indication establishes that indication as a terminal production of 'MODE mode indication' (7.2) or 'PRIORITY indication' (7.3). Monadic-indications have no indication-defining occurrence. }

4.3. Operators {24 July 1968}

{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). 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 : procedure with PARAMETERS ADIC operator{c,d}.

b) procedure with PARAMETERS MOID ADIC operator{75b,84b,g} :
   procedure with PARAMETERS ADIC operator{c,d}.

c) procedure with LMODE parameter and RMODE parameter
   PRIORITY operator{b} : PRIORITY indication{42e}.

d) procedure with RMODE parameter monadic operator{b} :
   monadic indication{42f}.

e)* priority operator :
   procedure with PARAMETERS PRIORITY operator{c}.

{Examples:

c) + ;

d) abs }

4.3.2. Identification of operators

{The identification of operators is similar to that of identifiers and indications, except that one same priority-indication may be more than one operator and therefore the modes of the operands must be considered. }

a) A given occurrence of an operator operator-defines if it precedes the equals-symbol of an operation-declaration {7.5}; otherwise, it is "operator-applied".

b) If a given occurrence of an operator is operator-applied, then it may identify an operator-defining occurrence of the operator 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, of an operator which is the same adic-indication as the given occurrence, and which is such that the mode of the left (right) operand of the operator can be firmly coerced to {4.4.3.a} the mode specified by the first (second) virtual-parameter in the plan of that operation-declaration {7.5.1.a,b}, then the given occurrence identifies that operator-defining occurrence of the operator; otherwise, Step 2 is taken.".

{Operators have no inherent meaning. The operator-defining occurrence of an operator is made to possess a routine (2.2.3.4) by the elaboration of an operation-declaration (7.5).

A given occurrence of an indication may be both a priority-indication and a priority-operator. As a priority-indication, it identifies its indication-defining occurrence. As a priority-operator, it may identify an operator-defining occurrence, which possesses a routine. Since the occurrence of an 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-operator is a subset of those occurrences which identify the same priority-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 occurrence of a 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 [prr]-priority-SIX-operator (4.3.1.b; i.e. ignoring the mode of the value, if any, which it delivers), where [prr] stands for procedure-with-real-parameter-and-real-parameter, and [prrr] for [prr]-real. The third occurrence of <u>min</u> is indication-applied and, as such, identifies the first occurrence, whereas, at the same textual position, <u>min</u> is also operator-applied, and, as such, identifies the second occurrence; this makes it (in view of Step 3) a [prr]-priority-SIX-operator and hence, because of the identification condition (4.4.1), a [prrr]-priority-SIX-operator. This identification of the priority-operator is made because:

i) <u>min</u> occurs in an operation-declaration,

ii) the base y can be firmly coerced to the mode specified by <u>real</u>,

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

iv) <u>min</u> is thus a [prr]-priority-SIX-operator, and

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

If the identification condition 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.}

```
  (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 indication and all of whose corresponding constituent virtual-parameters {7.5.1.b,7.1.1.x} are virtual-declarers specifying modes related to one another{4.4.3.b}.

{e.g., neither the closed-clause
```
 (op max = (int a, int b)int : (a > b | a | b) ;
  op max = (int a, int b)int : (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)int : (a > b | a | b) ;
  op max = (real a, real b)real : (a > b | a | b) ; sin(3.14))
```
may be.}

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.}

4.4. Context conditions {22 July 1968}

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 ~~must~~ 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, each applied occurrence of an identifier (each indication-applied occurrence of an indication, each operator-applied occurrence of an operator) which is a terminal production of one or more notions ending with 'identifier' ('indication', 'operator') is a terminal production of all those same notions at the defining (indication-defining, operator defining) occurrence~~, if any,~~ of that identifier (indication, operator). {See the remarks after 4.1.2 and 4.3.2, and for the significance of "one or more", see rule 4.3.1.b.}

b) No proper program contains an applied occurrence of an identifier (indication-applied occurrence of a mode-indication or priority-indication, operator-applied occurrence of an operator) which does not identify a defining (an indication-defining, an 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 occurrences of a given identifier nor two indication-defining occurrences of a given indication.

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

(real x, real x ; sin(3.14)),

c) No proper program contains a declarer {7.1} 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.}

d) No proper program contains a declarer 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 contained in an actual-declarer is "shielded" if

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

ii) it is contained in a virtual-parameter {7.1.1.y}, or

iii) it is contained in a virtual-declarer following a virtual-parameters-pack {5.4.1.i}.

{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 which is a mode-indication "shows" that mode-indication; an actual-declarer shows all mode-indications contained in it which are not shielded, and furthermore all mode-indications shown by the actual-declarer following the equals-symbol following the defining-occurrence of each such mode-indication.

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

c) No proper program contains a mode-declaration 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) }

5. Denotations {22 July 1968}

{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} ;
   row of boolean 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) A denotation possesses a value; a given denotation always possesses the same value; its elaboration involves no action.

b) The mode of the value possessed by a given denotation is obtained by deleting 'denotation' from that direct production of the notion 'denotation' of which the given denotation is a terminal production. {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,51a,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,512c,d,h,510b} : digit zero{303d} ;
    natural numeral{b}.

b) natural numeral{a} :
    digit FIGURE{303d}, digit token{303c} sequence option.

{Examples:
a) 0 ; 4096 ;
b) 1 ; 2 ; 3 ; 123 (Note that 00123 and -1 are not integral-denotations.)}

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 {22 July 1968}

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} ;
   integral part{c}, point symbol{31b}.
c) integral part{b} : integral denotation{511a}.
d) fractional part{b} : point symbol{31b},
   digit zero{303d} sequence option, 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 ; 123. ;
c)	123 ;	d)	.123 ; .000123 ;
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 divided 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

{Examples:
a) <u>true</u> ; <u>false</u> }

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 item{521b}, quote symbol{31i}.

{Examples:
a) "a" }

5.1.4.2. Semantics {22 July 1968}

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 character-token is a quote-image,} it is a new instance of the character possessed by the quote-symbol.

5.2. Row of boolean denotations

{There are two kinds of denotations of multiple values viz., bits, e.g. <u>1011</u>, 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) row of boolean denotation{860a} :
 long symbol{31d} sequence option, flipflop{303e} sequence.

{Examples:
a) <u>1011</u> ; <u>long</u> <u>1011</u> }

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 $\leq$ n, then the value of the row-of-boolean-denotation is 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 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).

{If the value of bits width is, say, 5, then <u>1011</u> possesses the same value as the collateral-clause (<u>false</u>, <u>true</u>, <u>false</u>, <u>true</u>, <u>true</u>), but <u>1011</u> is not a collateral-clause.}

5.3. Row of character denotations {24 July 1968}

{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 atring- 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},
 string item{c} sequence, 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.Step 4}.
d) quote image{c} : quote symbol{31i}, quote symbol{31i}.

{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 arrayed 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 {22 July 1968}

{A routine-denotation, e.g. (<u>real</u> a, b)<u>real</u> : (a > b | b | a), always has a routine-symbol (:). To the left of this symbol stand the formal-parameters, e.g. (<u>real</u> a, b), and a declarer specifying the mode of the value delivered, if any, e.g. <u>real</u>. To the right of the routine-symbol is the body, e.g. (a > b | b | a), which is always a closed-, conditional- or collateral-clause. If the routine delivers no value, then a void-symbol stands before the routine-symbol, e.g. <u>void</u> : <u>go to</u> princeton. In some cases this void-symbol may be omitted, see the extension 9.2. . It is essential that the body of a routine-denotation be CLOSED, for otherwise denotations like (<u>int</u> sintzoff)<u>void</u> : (<u>int</u> branquart)<u>void</u> : lewi (wodon) could also be clause calls, or formulas like (<u>int</u> a)<u>int</u> : 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} :
      formal procedure PARAMETY MOID plan{c,d},
      routine symbol{31b}, MOID body{h}.
c)  VICTAL procedure with PARAMETERS MOID plan{b,75b,71x} :
      VICTAL PARAMETERS{e,f,71y,74b} pack,
                               virtual MOID declarer{g,71b}.
d)  VICTAL procedure MOID plan{b,71x} :
      virtual MOID declarer{g,71b}.
e)  VICTAL PARAMETERS and PARAMETER{c,b,862a} :
      VICTAL PARAMETERS{e,f,71y,74b}, comma symbol{31e},
                              VICTAL PARAMETER{f,71y,74b}.
f)  formal MODE parameter{c,e,74a} :
      formal MODE declarer{71b}, MODE identifier{41b}.
g)  virtual void declarer{c,d} : void symbol{31b}.
i)* VICTAL parameters pack : VICTAL PARAMETERS{e,f,71y,74b} pack.
```

{Examples:

b) (**bool** a, b)**bool** : (a | b | **false**) ;
  void : (n = 1966 | warsaw | zandvoort) ;
c) (**bool** a, b)**bool** ;
d) **void** ;
e) **bool** a, **bool** b ;
f) **bool** a ;
g) **void** ;
h) (a | b | **false**) ; (n = 1966 | warsaw | zandvoort)}

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 a void-symbol, then that void-symbol and the routine-symbol which follows it is deleted and Step 4 is taken; otherwise, the routine-symbol is replaced by a becomes-symbol, and a value-of-symbol followed by an open-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 = **void** : (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)$\overset{\text{real}}{\wedge}$: (a > 0 | xx | yy), is (**val**(**int** a = **skip** ; **real** := (a > 0 | xx | yy))), and that possessed by p4 after the elaboration of <u>proc</u> p4 = (<u>real</u> a, b) : (a > b | stop) is (<u>real</u> a = <u>skip</u>, <u>real</u> b = <u>skip</u> ; (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 (clause-calls). }

5.5. Format denotations

5.5.1. Syntax

a) format denotation :
 formatter symbol, collection list, formatter symbol.

b) collection : picture ; insertion option, replicator,
 collection list pack, insertion option.

c) picture : MODE pattern, insertion option.

d) insertion : literal option, insert sequence ; literal.

e) insert : replicator, alignment, literal option.

f) replicator : replication option.

g) replication : dynamic replication ; integral denotation.

h) dynamic replication : letter n, fitted serial integral expression pack.

i) alignment : letter k ; letter x ; letter y ; letter l ; letter p.

j) literal : STRING denotation option , replicated literal sequence ;
 STRING denotation.

k) replicated literal : replication, STRING denotation.

{Examples:

a) *fp"table.of"x10a,n(lim-1)(l6x3zd,3x10(2x+.12de+2d"+j×"si+.10de+2d))pf* ;

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

c) *l20kc("mon","tues","wednes","thurs","fri","satur","sun")"day"* ;

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 : loose replicatable zero frame, sign frame; loose sign frame.

m) loose ANY frame : insertion option, ANY frame.

n) replicatable ANY frame : replicator, ANY frame.

o) zero frame : letter z.

p) sign frame : plusminus.

q) suppressible ANY frame : letter s option, ANY frame.

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 declarations, viz.

i) formatless transput (10.5.2)

ii) formatted transput (10.5.3)

iii) binary transput (10.5.4.)

Formats (see 5.5.2.a) are used by the formatted transput routines to control input from and output to a "file" (10.5.1).

bb) A format may be associated with a file by *format* (10.5.1.3.a), thereby causing its first constituent picture to be the current picture of the file.

After the current picture of the file has been used to control the transput of a value, then, unless the format is exhausted, the next picture of the format is made to be the current picture of the file. If no format has been associated with the file, or if the format associated with the file is exhausted, then the current picture of the file is undefined.

cc) The current picture of the file is used on output to control the "conversion" of a value to a "string", i.e. a value of mode 'row of character', and, on input, that of a string to a value.

dd) The mode specified by a picture is that obtained by deleting 'pattern' from that notion ending with 'pattern' whose terminal production is the constituent pattern of that picture.

ee) Formats have a complementary meaning on input and output; that is, under control of one given picture:

i) it is possible to convert a given value to a string by means of a formatted output routine, provided the mode specified by the picture is "output-compatible" with the mode of the given value, and the number of characters specified by the picture is sufficient (10.5.3.1);

ii) it is possible to convert a given string to a value of a given mode, provided the mode specified by the picture is "input-compatible" with the mode of the value, the number of elements of the string is the same as that specified by the picture, and the individual characters of the string "agree" with the frames of the picture specifying them (10.5.3.2);

iii) if it is possible to convert a given value to a string and 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 it is possible to convert the resulting string (under control of the same picture) into a value; the resulting value is equal (approximately equal) to the given value if the given value is a string, integer or truth value (is a real value) ;

iv) if it is possible to convert a given value into a string and to convert that string into a new value, then converting this new value to a string yields the same string.

ff) The value of the empty replicator is one; the value of a replication that is an integral-denotation is the value of that denotation; the value of a dynamic-replication is the value of its constituent serial-integral-expression if that value is positive, and zero otherwise.

gg) The number of characters specified by a picture is the sum of the numbers of characters specified by its constituent frames and the number specified by a frame is equal to the value of its preceding replicator.

hh) A frame preceded by letter-s is "suppressed", and the characters specified by it are also suppressed, i.e.:
on output, are deleted from the string that is output, and,
on input, are inserted in the string that is input, viz., by inserting the character possessed by a point (times-ten-to-the-power, plus-i-times, digit-zero, space) -symbol for a suppressed-point (exponent, complex, digit, character) -frame.

ii) 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.i, j, k) some character is "present", depending on the contents of the file and on its "conversion-string".

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

kk) Performing an alignment affects the position of the file as follows, where n stand 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.m) ;
c) letter-y causes the char number to be decremented by n (10.5.1.2.n) ;

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.o) ;

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.p).

ll) A (replicated-) literal is written by writing the string possessed by its constituent row-of-character-denotation (as many times as 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.

mm) A (replicated-) literal is required by requiring the string possessed by its constituent row-of-character-denotation (as many times as 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.

nn) When a string is "read" whose number of characters is given, 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.

oo) A picture can be used to "edit" a value as follows:

i) The value is converted by an appropriate output routine (10.5.2.c, d, e) to a string of as many characters as specified by the picture. If the picture is or contains an integral-pattern or real-pattern, then this conversion takes place to a base equal to the radix, if present, and base ten otherwise.

ii) If the picture contains a sign-mould, then 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 (negative), then a space (minus), and, otherwise, a plus (minus). This character is shifted in that part of the string specified by the sign-mould as far to the right as possible across any leading zeroes and those zeroes are replaced by spaces; e.g., under the sign-mould *4z+*, the string possessed by *"+0003"* is edited into that possessed by *"*$\underline{.}\underline{.}\underline{.}$*+3"*. If the picture does not contain a sign-mould and the value is negative, then the result is undefined.

iii) 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* is edited into the string possessed by *"18*$\underline{.}$*168"*.

iv) Suppressed characters are deleted.

pp) A picture can be used to "indit" a string into a value of a given mode as follows:

i) If the picture contains a sign-mould, then the character specified by its constituent sign-frame must be one of the characters specified by that sign-mould. Only spaces may appear in front of this character and no leading zeroes may appear after it. The leading spaces are deleted, and if the character specified by the sign-frame is a space, and the sign frame is a minus-symbol, then that character is replaced by a plus.

ii) Leading spaces in those parts of the string specified by any remaining zero-frame are replaced by zeroes.

iii) For each suppressed digit, a zero is inserted into the string; for each other suppressed character, a space is inserted.

iv) The string is converted by an appropriate input routine (10.5.3.b, c, d) into a value of the given mode.

qq) The *insertion*, if any, following the constituent *pattern* of a *picture* is performed after the *pattern* has been used.

The insertion, if any, preceding the constituent collection-list-pack of a collection that is not a picture is performed before the first constituent picture is used to control the transput of a value. The insertion, if any, following that collection-list-pack is performed after all constituent pictures have been used. }

5.5.1.1. Integral patterns

a) integral pattern : radix mould option, sign mould option, integral mould ; integral choice pattern.

b) radix mould : insertion option, radix, letter r.

c) radix : digit two ; digit four ; digit eight ; digit one, digit zero ; digit one, digit six.

d) integral mould : loose replicatable suppressible digit frame sequence.

e) digit frame : zero frame ; letter d.

f) integral choice pattern : insertion option, letter c, literal list pack.

{Examples:

a) *2r6d30sd* ; *12z+d* ; *zd"-"zd"-19"2d* ;
l20kc("mon","tues","wednes","thurs","fri","satur","sun") ;

b) *2r* ;

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

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

f) *l20kc("mon","tues","wednes","thurs","fri","satur","sun")* }

{If the integral-pattern is not an integral-choice-pattern, then,

i) on output, the value to be output is edited into a string and "transcribed onto" the file by, for all frames occurring in the pattern, first performing the preceding insertion, if any, and then writing on the file that part of the string specified by the frame;

ii) on input, a string is "transcribed from" the file, which string is obtained by, for all frames occurring in the pattern, first performing the preceding insertion, if any, and then, for a frame that is not suppressed, reading from the file as many characters as are specified by the frame ; that string is indited into a value.

If the integral-pattern is an integral-choice-pattern, then the insertion, if any, preceding the letter-c is performed, and,

i) 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;

ii) 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 .}

5.5.1.2. Real patterns

a) real pattern : sign mould option, real mould ; floating point mould.

b) real mould : integral mould, loose suppressible point frame, integral mould option ; loose suppressible point frame, integral mould.

c) point frame : point symbol.

d) floating point mould : stagnant mould, loose suppressible exponent frame, sign mould option, integral mould.

e) stagnant mould : sign mould option, INTREAL mould.

f) exponent frame : letter e.

{Examples:

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

b) *d.11d* ; *.12d* ;

d) *+d.11de+2d* ;

e) *+d.11d* }

{On output, under control of a real-pattern, a real or integral value is edited into a string and transcribed onto the file; on input, a string is transcribed from the file and indited into a real value. }

5.5.1.3. Boolean patterns

a) boolean pattern : insertion option, letter b, boolean choice mould option.

b) boolean choice mould : open symbol, literal, comma symbol, literal, close symbol.

{Examples:

a) *l"result"l4xb* ; *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,

i) 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;

5.5.1.3. continued

ii) 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

5.5.1.4. Complex patterns

a) COMPLEX pattern :
real pattern, loose suppressible complex frame, real pattern.

b) complex frame : letter i.

{Example:

a) *2x+.12de+2d"+j×"si+.10de+2d* }

{On output, the complex or real or integral value is edited into a string and transcribed onto the file; on input, a string is transcribed from the file and indited into a complex value. }

5.5.1.5. String patterns

a) STRING pattern : loose string frame;
loose replicatable suppressible character frame sequence.

b) string frame : letter t.

c) character frame : letter a.

{Example:

a) *p"table_of"x10a* }

{If the pattern is a loose-string-frame then the insertion, if any, preceding its constituent letter-t is performed, and,

i) on output, the given string is written on the file ;

ii) on input, if the string has fixed bounds, then that number of characters are read, otherwise a string is read under control of the terminator-string referenced by the file (10.5.1.mm);

otherwise,

i) on output, the given string, which must have as many elements as the number of characters specified by the format-item, is edited into a string and transcribed onto the file ;

ii) on input, a string is transcribed from the file and indited into a string.

If the value to be transput is a character, then a string having that character as its only element is transput.}

5.5.1.6. Transformats

a) structured with a STRING named letter aleph~~x~~ transformat :
strong unitary format clause.

{Example: *(x≥0|f5df|f5d"-"f)* }
{For unitary-clauses see Chapter 8.}

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

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

5.5.2. 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: It is replaced by the format obtained in Step 1, and the thereby resulting format-denotation is considered ;
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 that 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 would be obtained by replacing, in the considered format-denotation as modified in Step 4, each constituent quote-symbol by a quote-image {5.3.1.c} 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 considered format-denotation is replaced by the given transformat, and that transformat is made to possess the structured value obtained in Step 6.

6. Phrases {19 July 1968}

{A phrase is a declaration or a clause. Declarations may be unitary, e.g. <u>real</u> x, or collateral, e.g. <u>real</u> 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. <u>if</u> x > 0 <u>then</u> x <u>else</u> 0 <u>fi</u> (which may also be written (x > 0 | x | 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)* phrase : SORTETY SOME PHRASE{61a,62a,b,c,e,64d,e,70a,81a}.
b)* SOME phrase : SORTETY SOME PHRASE.
c)* statement : strong void unit{61e}.

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", 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 an occurrence of an identifier (indication) which is the same as some identifier (indication) occurring outside the given clause defines {4.1.2.a} (indication-defines {4.2.2.a}) within it, then the defining (indication-defining) occurrence and all occurrences identifying it are replaced by occurrences of one same identifier (indication) which does not occur elsewhere in the program and Step 1 is taken; otherwise, Step 2 is taken ;

Step 2: If an occurrence of an indication which is the same as some indication occurring outside the given clause is operator-defined within it, then the operator-defining occurrence and all occurrences identifying it are replaced by occurrences of one same new indication which does not occur elsewhere in the program and Step 3 is taken; otherwise, the protection of the given clause is complete ;

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

Step 4: A copy is made of the priority-declaration containing that occurrence of the indication which, before the replacement in Step 2, was identified by that operator; the occurrence of that indication in the copy is replaced by an occurrence of that new indication; the copy, thus modified, preceded by an open-symbol and followed by a go-on-symbol, is inserted preceding 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 {17 July 1968}

{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 x1 ; for i to n do x1[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,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) STRONG MOID unit{d,i,55h,62b,c,h,71s,74b,831f,861h,j,k} :
STRONG 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 CLAUSE trains{a,g} : FIRM CLAUSE train{h} ;
FIRM CLAUSE train{h}, completer{l},
suite of strong CLAUSE trains{f} ;

```
      coFIRM CLAUSE train{h}, completer{l},
                              suite of FIRM CLAUSE trains{g}.
h)  SORTETY CLAUSE train{f,g} : label{k} sequence option,
      statement prelude{i} option, SORTETY unitary CLAUSE{81a}.
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 ; (x > 0 | l3 | x := 1 - x) ; (p | l3) ; *
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-clause or declaration.

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

c) The elaboration of a serial-clause is
i) interrupted (halted, resumed) upon the interruption (halting, resumption) of a constituent unitary-clause or declaration ;
ii) terminated upon the termination of the elaboration of a constituent unitary-clause or declaration appointing a successor outside the serial-clause, and that successor {6.2.2.b} is appointed the successor of the serial-clause.

c) The eleboration 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 eleboration 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 := (<u>real</u> r := 0.1 ; r), the value of the serial-clause <u>real</u> r := 0.1 ; r is undefined since the scope of the name possessed by r is the serial-clause itself, whereas, in y := (<u>real</u> r := 0.1; r), the serial-clause <u>real</u> r := 0.1 ; r possesses a real value.}

6.2. Collateral phrases {23 July 1968}

{Collateral-phrases contain two or more unitary-phrases separated by comma symbols (, or <u>comma</u>) and, in the case of collateral clauses, are enclosed between an open ((or <u>begin</u>) and a close () or <u>end</u>), e.g. (x := 1, y := 2) or <u>real</u> x, <u>real</u> y (usually <u>real</u> 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 []<u>real</u> x1 = (1.2, 3.4) and in <u>compl</u> 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 <u>struct</u> <u>m</u> = (<u>ref</u> <u>m</u> m) ; <u>m</u> nobuo, yoneda, the assignation nobuo := (yoneda) would be ambiguous. In the range of <u>struct</u> <u>r</u> = (<u>real</u> a) ; <u>r</u> r, the construction r := (3.14) is not an assignation, but a <u>of</u> r := 3.14 is. It is possible to present a single value or no value at all as a multiple value, e.g. []<u>real</u> x1 := ; []<u>real</u> y1 := 3, but this involves a coercion known as arraying, see 8.2.6.}

6.2.1. Syntax

```
a)  collateral declaration{61c} :
      unitary declaration{70a} array,
b)  STRONG collateral void clause{81d} :
      parallel symbol{31e} option,
               STRONG void unit{61e} array box.
c)  STRONG collateral REFETY row of MODE clause{81d} :
      parallel symbol{31e} option,
               STRONG MODE unit{61e} array box.
d)  FIRM collateral row of MODE clause{81d} :
      parallel symbol{31e} option, FIRM MODE balance{e} box.
e)  FIRM MODE balance{c} :
      FIRM MODE unit{61e}, comma symbol{31b},
               STRONG MODE unit{61e} list ;
      coFIRM MODE unit{61e}, comma symbol{31b},
               FIRM MODE unit{61e}.
      coFIRM MODE unit{61e}, comma symbol{31b}
               FIRM MODE balance{e}.
```

f) STRONG collateral REFETY structured with FIELDS and FIELD clause{81d} : parallel symbol{31e} option ; STRONG structured with FIELDS and FIELD structure{g} box.

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) <u>real</u> x, <u>real</u> y ; (and by 9.2.c) <u>real</u> x, y ;

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

c) (x, n) ;

d) (1, 2) (in []<u>real</u> x1 := (1, 2) ;

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

i) initiated by initiating the elaboration of each of these constituents ;

ii) interrupted upon the interruption of the elaboration of any of thses constituents ;

iii) completed upon the completion of the elaboration of all of these constituents; and

iv) 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: Either 'void' or a mode is obtained by deleting 'collateral', 'clause' and the terminal production of 'STRONG' from {the notion which is} that direct production of 'collateral clause' of which the given collateral-clause is a terminal production, and if this is 'void', then the elaboration of the collateral-clause is complete; otherwise, Step 3 is taken ;
Step 3: The mode obtained in Step 2 is possibly modified by deleting from it an initial 'reference to', if any, and the mode then obtained is considered ;
Step 4: If the mode considered in Step 3 begins with 'row of', then Step 5 is taken; otherwise, 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} whose mode is that considered mode; this structured value is considered and Step 7 is taken ;
Step 5: 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 6 is taken ;
Step 6: A new instance of a multiple value, whose mode is that considered in Step 3, 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 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 $i = n, n-1, \ldots ,2$, 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 7 is taken ;

Step 7: If the mode obtained in Step 2 {, not that considered in Step 3,} does not begin with 'reference to', then the value of the collateral-clause is the considered value; otherwise, a name, different from all other names, whose scope is the program and whose mode is that obtained in Step 2, is created; this new name, which is then made to refer to the considered value, is the value of the collateral-clause.

6.3. Closed clauses

{23 July 1968}

{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{54h,81d} :
      elementary symbol{31e} option,
                                  SORTETY serial CLAUSE{61a} box.
```

{Examples:

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

6.3.2. Semantics

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

b) The elaboration of a closed-clause which begins with an elementary-symbol is an elementary action {2.2.5}.

6.4. Conditional clauses {17 July 1968}

{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)  FIRM choice CLAUSE{a} :
       FIRM then CLAUSE{e}, strong else CLAUSE{e} option ;
       coFIRM then CLAUSE{e}, FIRM 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 then 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 cofirm and | y firm. The example (p | skip | skip) + y illustrates that not both may be strong, for otherwise the operator + could not be identified. Cofirm, which means strong but not firm, is used merely to avoid ambiguous parsing. }

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

i) interrupted (halted, resumed) upon the interruption (halting, resumption) of the elaboration of the condition or the considered clause ;

ii) completed upon the completion of the elaboration of the considered clause, if any; otherwise, completed upon the completion of the elaboration of the condition ;

iii) 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 {23 July 1968}

{Unitary-declarations provide the defining uccurrences for mode-indications, e.g. string in mode string = [1:]char, priority indications, e.g. plus in priority plus = 1, identifiers, e.g. x in real x, and operators, e.g. abs in op abs = (int a) int : (a < 0 | -a | a). Declarations appear 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 bits = [1:bits width]bool ; priority plus = 1 ;
    int m = 4096 ; op ÷ = (real a, b)int : (round a ÷ round b)}

7.0.2. Semantics

An external object {2.2.1} which was caused to possess a value by the elaboration of a declaration is caused to possess an undefined value upon termination or completion of the elaboration of the smallest range containing that declaration.

7.1. Declarers {23 July 1968}

{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, e.g. []**real**, [1:n]**real** and [1:**int** n]**real** are all formal, but only the first two are actual and only the first is virtual.}

7.1.1. Syntax

a)* declarer : VICTAL MODE declarer{b}.
b) VICTAL MODE declarer{1,m,n,o,y,54c,d,f,851b,c} :
    VICTAL MODE declarator{c,d,e,k,1,m,n,o,x,aa} ;
    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) VIRACT structured with FIELDS declarator{b} :
    structure symbol{31d}, VIRACT FIELDS declarator{f,h} pack.
f) VIRACT FIELDS and FIELD declarator{e,f,k} :
    VIRACT FIELDS declarator{f,h}, comma symbol{31e},
        VIRACT FIELD declarator{h}.
g)* field declarator : VIRACT FIELD declarator{h}.
h) VIRACT MODE named TAG declarator{e,f,h} :
    VIRACT MODE declarer{b}, MODE named TAG selector{j}.
i)* field selector : FIELD selector{j}.

j) MODE named TAG selector{852a,g} : TAG{302b,41c,d}.
k) formal structured with FIELDS declarator{b} :
    structure symbol{31d}, virtual FIELDS declarator{f,h} pack.

{Examples:
e) **struct**(**string** name, **real** value) ;
f) **string** name, **real** value ;
h) **string** name ;
j) name ;
k) **struct**(**string** name, []**real** value) }

{Rule h, together with 1.2.1.k.l.m.n.o.p 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 (8.5.3). 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 range does not cause any ambiguity. Thus, **mode** **cell** = **struct**(**string** name, **ref** **cell** next) and **mode** **link** = **struct**(**ref** **link** next, **ref** **cell** value) may both be present in the same range. }

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

{Examples:
l) **ref**[]**real** ;

m) ref[1:int n]real ; ref[]real ;
n) ref ref[]real }

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

o) VICTAL ROWS NONROW declarator{b} : sub symbol{31e},
VICTAL ROWS rower{p,q}, bus symbol{31e},
virtual NONROW declarer{b}.
p) VICTAL row of ROWS rower{o,p} :
VICTAL row of rower{p}, comma symbol{31e},
VICTAL ROWS rower{p,q}.
q) VICTAL row of rower{o,p} :
VICTAL lower bound{r,s,u}, up to symbol{31e},
VICTAL upper bound{r,s,u}.
r) virtual LOWPER bound{q} : EMPTY.
s) actual LOWPER bound{q,861f} :
strict LOWPER bound{t} option.
t) strict LOWPER bound{s,u} : strong integral unit{61e}.
u) formal LOWPER bound{q} : strict LOWPER bound{t} option.
LOWPER bound interrogation{v} option,
LOWPER state interrogation{w}.
v) LOWPER bound interrogation{u} :
integral symbol{31d}, integral identifier{41b}.
w) LOWPER state interrogation{u} : true symbol{31b} option ;
false symbol{31b} ; boolean symbol{31d},
boolean identifier{41b}.

{Examples:
o) [1:m,1:n] ;
p) 1:m,1:n ;
q) ; 1:m ; 1:int n ;
s) m ; ;
t) m ;
u) m ; int m ; bool s ; int n false ; 10 false ;
v) int n ;
w) true ; ; false ; bool s }

{Upper-bound-interrogations, e.g. int n, are used to ask for the value of upper bounds in actual-parameters, e.g. [1:int n]real x1 = (1.2, 3.4), will result in n possessing the value of 2. Upper-state-interrogations, e.g. bool b, are used to ask for or prescribe the value of upper states in actual-parameters, e.g. [1:bool b]real x1 = (1.2, 3.4) will result in b possessing the value true, corresponding to an upper state 1 (i.e., the upper bound may not vary) in the descriptor of (1.2, 3.4). An upper-state-interrogation like that in [1:int n false] char s = t, may be used to prescribe that only those multiple values with upper state 0 (i.e. the upper bound may vary) may be possessed by s, while that in [1:10]char s = t, or in [1:10 true]char s = t, may be used to prescribe that only those multiple values with upper state 1 may be possessed by s. Similar remarks apply to lower-bound-interrogations an lower-state-interrogations.}

x) VICTAL PROCEDURE declarator{b} :
    procedure symbol{31d}, virtual PROCEDURE plan{54c,d}.
y) virtual MODE parameter{54c,e} : virtual MODE declarer{b}.
z) parameters pack : VICTAL PARAMETERS{y,54e,f,74b} pack.

[illegible]
x) proc ; proc(real, int) ; proc(real)bool ;
y) real }

aa) VICTAL union of MODES mode declarator{b} :
    union of symbol{31d}, virtual MODES declarer{b} pack.
ab) virtual MODES and MODE declarer{aa} :
    virtual MODES declarer{ab,b}, comma symbol{31e},
        virtual MODE declarer{b}.

{Examples:
aa) union(int, bool) ;
ab) int, bool }

7.1.2. Semantics {24 July 1968}

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" if 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}; if the copy {in its new position} is now a formal-declarer, then for each lower-state-interrogation and upper-state interrogation in the copy, if any, which is empty, a false-symbol is inserted and the Step is taken again; otherwise, the development of the declarer has been accomplished. {Thus, e.g. the elaboration of **string** s := "a", results in **ref**[1:**false**]**char** s = **loc**[1:]**char** := "a". }

{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 and s_i (t_i) to 1, and otherwise s_i (t_i) is set to 0 {and l_i (u_i) is undefined}; 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 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 defining occurrences of mode-indications, which act as abbreviations for declarers built from primitive-symbols, e.g. <u>mode</u> <u>string</u> = [1:]<u>char</u>, or from other declarers or even from themselves, e.g. <u>mode</u> <u>book</u> = <u>struct</u>(<u>string</u> title, <u>ref</u> <u>book</u> next). In this last example, the mode-indication is not only a convenient abbreviation but it is essential to the declaration. }

7.2.1. Syntax

{23 July 1968}

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

{Examples:

```
a) mode bits = [1:bits width]bool ;
     struct compl = (real re, im) (se 9.2.b,c) ;
     union primitive = (int, real, bool, 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 defining occurrences for priority-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}.

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 identifiers, e.g. x in **real** x (which is an abbreviation of **ref** **real** x = **loc** **real**, see 9.2.a). Their elaboration causes identifiers to possess values; in the 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{54f}, 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{5516a}.

{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 ;

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

```
    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 constituent strict-lower-bounds and strict-upper-bounds of 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 value considered in Step 2 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 ;

Step 4: Each defining occurrence, if any, of an identifier in a constituent lower-bound-interrogation (lower-state-interrogation) or upper-bound-interrogation (upper-state-interrogation) of the formal-parameter is made to possess a new instance of the corresponding bound (of true, if the corresponding state is 1, and of false otherwise) in the value considered in Step 2 ;

Step 5: All applied occurrences of identifiers in constituent lower-bound-interrogations (lower-state-interrogations) and upper-bound-interrogations (upper-state-interrogations) of that formal-parameter are elaborated collaterally; for each constituent lower-state-interrogation and upper-state-interrogation, if the value possessed by its constituent identifier is false (true) and the corresponding state in the value considered in Step 2 is 1 (0) or if it is a true-symbol-option (false-symbol) and the corresponding state in the value considered in Step 2 is 1 (0), then the further elaboration is undefined; for each constituent lower-bound-interogation and upper-bound-interrogation, if the value possessed by its constituent identifier is not the same as that of the corresponding bound in the value considered in Step 2, then the further elaboration is undefined; otherwise, the identifier of the formal-parameter is made to possess a new instance of the value of the actual-parameter.

{According to Step 5, the elaboration of the declarations [1:2]<u>real</u> x1 = (1.2, 3.4, 5.6) and [1:<u>int</u> n, 1:<u>int</u> n]<u>real</u> x2 = ((1.1, 1.2), (2.1, 2.2), (3.1, 3.2)) is undefined. Similarly, in the range of <u>string</u> s1 = "abc", s2 = "def", the elaboration of [1:<u>true</u>]<u>char</u> r = s1 and [1:<u>bool</u> s, 1:<u>bool</u> s]<u>char</u> t = (s1, s2) is undefined. }

7.5. Operation declarations {23 July 1968}

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

7.5.1. Syntax

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

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

{Examples:

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

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.}

{Unitary-clauses may appear as actual-parameters, e.g. x in sin(x), as sources in assignations, e.g. y in x := y, 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 on the left of assignations, or may be in identity- or conformity-relations, or may be strict-lower- or strict-upper-bounds, e.g. x in x + 1 or in x := 1 or in x :=: yy or in x ::= ir or n and m in [n:m]<u>real</u> x1; 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,h} :
       SORTETY MOID tertiary{b} ;
       SORTETY MOID confrontation{820d,e,f,g,h,i,j,830a}.
b)  SORTETY MOID tertiary{a,831e,832a,833a} :
       SORTETY MOID secondary{c} ;
       SORTETY MOID ADIC formula{820d,e,f,g,h,i,j,84b,g}.
c)  SORTETY MOID secondary{b,84f,852a} :
       SORTETY MOID primary{d} ;
       SORTETY MOID cohesion{820d,e,f,g,h,i,j,850a}.
```

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

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

8.2. Coercends {18 July 1968}

{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, arraying, 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'; in x := 1 the base 1, whose a priori mode is 'integral', is widened to 'real'; in string s := "a" the base "a", which is of a priori mode 'character', is arrayed to 'row of character'; in x := skip the skip 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 x
(i.e. the modes before and after coercion). Four sorts of syntactic positions may be described: "strong" positions, i.e. act-

ual-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, (operands are the only firm positions); "weak" positions, i.e. certain primaries, e.g. sin in sin(x) and x1 in x1[i] and certain secondaries, e.g. z in re <u>of</u> z; and "soft" positions. i.e. destinations, e.g. x in x := y and certain tertiaries, e.g. xx in xx :=: x.

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 the operands, in which widening, arraying, 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 + <u>skip</u> 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 remove a 'reference to' which precedes a 'NONREF' 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, etc.) of position is passed on, or modified during balancing (to cofirm, coweak or cosoft) 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), then 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 {18 July 1968}

```
a)* coercend : SORTETY COERCEND{d,e,f,g,h,i,j,830a,84b,g,850a,860a};
      SORTly ADAPTED to COERCEND{821a,b,822a,b,c,823a,824a,
                                   825a,b,826a,827a,828a,b}.
b)* SORT coercend : SORT COERCEND{d,e,f,g,h,i,j}.
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,
                                             826a,827a,828a,b}.
e) firm COERCEND : 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}.
h)  cofirm COERCEND{81a,b,c,d} :
       strongly widened to COERCEND{825a,b} ;
       strongly arrayed to COERCEND{826a} ;
       strongly hipped to COERCEND{827a} ;
       strongly voided to COERCEND{828a}.
i)  coweak COERCEND{81a,b,c,d} :
       strongly dereferenced to COERCEND{821a} ;
       strongly widened to COERCEND{825a,b} ;
       strongly arrayed to COERCEND{826a} ;
       strongly hipped to COERCEND{827a}.
j)  cosoft COERCEND{81a,b,c,d} :
       strongly dereferenced to COERCEND{821a} ;
       strongly hipped to COERCEND{827a}.
```

```
     {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]) ; z (in re of z)
g)  x (in x := 1) ; xory (in xory := 3.14) ;
h)  1 (in x + (x > 0 | 1 | y)) ;
i)  1 (in re of (x > 0 | 1 | z) ;
j)  xx (in (x > 0 | xx | x) := 3.14) }
```

8.2.1. Dereferenced coercends {18 July 1968}

{Coercends are dereferenced when it is required that the a priori mode should be changed by removing an initial 'reference to', 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,i,j,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 NONREF FORM{b,820f} :
 reference to reference to NONREF FORM{830a,84b,g,850a,860a};
 weakly FITTED to reference to reference to NONREF 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 range of ref[1:n]real rx1) }

8.2.1.2. Semantics

A dereferenced-coercend is elaborated in the following steps:
Step 1: It is preelaborated {1.1.6.f} ;
Step 2: If the value obtained in Step 1 is not nil, then the value of the dereferenced-coercend is a new instance 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 {18 July 1968}

{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 FORM{a,820d,e,821a,824a,825a,b, 826a,828b} : procedure MOID FORM{830a,84b,g,850a,860a} ; STIRMly FITTED to procedure MOID FORM{a,821a}.
b) weakly deprocedured to MODE FORM{820f,821b} : procedure MODE FORM{830a,84b,g,850a,860a} ; firmly FITTED to procedure MODE FORM{a,821a}.
c) softly deprocedured to MODE FORM{c,820g} : procedure MODE FORM{830a,84b,g,850a,860a} ; softly deprocedured to procedure MODE FORM{c}.

{Examples:
a) random (in x := random or in x + random) ;
b) rz (in re **of** rz in the range of **proc** **compl** rz = (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.f} 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 clause-calls, 8.6.2.}

8.2.3. Procedured coercends {19 July 1968}

{A coercend is procedured when it is required that the a priori mode should be altered by placing an initial 'procedure' before it (i.e. it should be turned into a procedure without parameters), e.g. x := 1 in **proc** **real** p := x := 1. However, special care must be taken with procedures which deliver no value, in order that clauses like (**proc** 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 (**proc** 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 FORM{a} :
   NONPROC FORM{830a,84b,g,850a,860a}.

{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) x := 1 (in **proc** p := x := 1) }

8.2.3.2. Semantics {19 July 1968}

A procedured-coercend is elaborated in the following steps:

Step 1: A copy is made of it {itself, not its value} and a routine-symbol followed by an open-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 'procedure void', then a void-symbol is placed before the copy and Step 3 is taken; otherwise, the initial 'procedure' is deleted from the considered mode and a virtual-declarer specifying the mode so obtained is placed before 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.

8.2.4. United coercends

{Coercends are united when it is required that the a priori mode should be changed to a united mode containing that mode, 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'.}

8.2.4.1. Syntax

a) STIRMly united to union of LMODESETY MODE RMODESETY FORM {a,820d,e,823a} : MODE FORM{830a,84b,g,850a,860a} ; firmly ADJUSTED to MODE FORM{a,821a,822a,823a}.

{Examples:

a) 2 ; i (in (union(int, real) ir ; ir := 2 ; ir := i)) }

{In rule a, '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 {18 July 1968}

{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. Widening for length numbers other than one is also provided. }

8.2.5.1. Syntax

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

{Examples:
a) 1 (in x := 1) ; i (in x := i) ;
b) 3.14 (in z := 3.14) ; x (in x := x) ; 1 (in z := 1) }

8.2.5.2. Semantics

A widened-coercend is elaborated in the following steps:

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

Step 2: If the value yielded in Step 1 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, it is a new instance of that structured {complex (10.2.5)} value composed of two fields, whose field-selectors are letter-r-letter-e and letter-i-letter-m, whose modes are the same as that of the value yielded in Step 1, and which are new instances of that value and zero respectively; its mode is that obtained by deleting 'strongly widened to' and the terminal production of 'FORM' from that notion as terminal production of which the widened-coercend is elaborated.

{Widening may not be done in firm positions, for otherwise i + 1 might be ambiguous. }

8.2.6. Arrayed coercends {19 July 1968}

{Coercends are arrayed when it is required that the a priori mode should be altered by placing an initial 'row of' before it, e.g. in [1:]real x1 := 3.14, the a priori mode of the base 3.14 is 'real' but the a post$\overset{er}{\wedge}$iori mode required in this strong position is 'row of real'. Here 3.14 is turned into a multiple value with a descriptor. }

8.2.6.1. Syntax

```
a)  strongly arrayed to REFETY row of MODE FORM{a,820d,h,i,823a}:
      MODE FORM{830a,84b,g,850a,860a} ;
      strongly ADJUSTED to MODE FORM{821a,822a,823a,824a} ;
      strongly widened to MODE FORM{825a,b} ;
      strongly arrayed to MODE FORM{a} ;
      MODE FORM vacuum{b}.
b)  NONROW base vacuum{a} : EMPTY.
```

{Examples:

```
a)  3.14 (in [1:]real x1 := 3.14) ; y (in [1:]real x1 := y) ;
      3.14 (in [1:]compl z := 3.14) ;
      3.14 (in [1:,1:]real x2 := 3.14) ;
      (the EMPTY in [1:]real := ) }
               ↑following :=
```

8.2.6.2. Semantics

An arrayed-coercend is elaborated in the following steps:

Step 1: If it is not empty, then it is preelaborated, 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 6 is taken ;

Step 3: If the value obtained in Step 1 is a name, then the value referred to by this name, and, otherwise the value obtained in Step 1 is considered; if the considered value is a multiple value, then Step 5 is taken ;

Step 4: 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 6 is taken ;

Step 5: A new instance of a multiple value is created, 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, and this new multiple value is considered instead ;

Step 6: The mode of the considered value is that obtained by deleting 'arrayed', the initial 'reference to', if any, and the terminal production of 'FORM' from that notion as terminal production of which the arrayed-coercend is elaborated; if that notion begins with 'arrayed row of', then the value of the arrayed-coercend is the considered value; otherwise, a name different from all other names, whose scope is the program and whose mode is 'reference to' followed by the mode of the considered value is created and made to refer to the considered value, and this name is then the value of the arrayed-coercend.

8.2.7. Hipped coercends

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

8.2.7.1. Syntax {19 July 1968}

a) strongly hipped to MOID base{820d,h,i,j} :
   MOID hop{b} ; MOID nihil{e,-}.
b) MOID hop{a} : skip{c} ; jump{d}.
c) skip{b} : skip symbol{31g}.
d) jump{b} : go to symbol{31f} option, label identifier{41b}.
e) reference to MODE nihil{a} : nil symbol{31g}.

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

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 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.h} 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 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 a routine-symbol followed by an open-symbol is placed before and a close-symbol is placed after the copy; if the considered mode is 'procedure void', then a void-symbol is placed before the copy; otherwise, the initial 'procedure' is deleted from the considered mode and a virtual declarer specifying the mode so obtained is placed before the copy; the value of the jump is the routine possessed by the {routine-denotation which is 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 'nihil' from that notion ending with 'nihil' of which the nihil is a terminal production.

{Skips play a role in the semantics of routine-denotations (5.4.2.Step 2) and clause-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) or unit (6.1.1.e) whose value is irrelevant or is to be calculated later; e.g. f(3, <u>skip</u>) 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 unitary-clause of a collateral-clause, e.g. [1:]<u>real</u> x1 := (3.14, <u>skip</u>, 1.68, <u>skip</u>) ;

iii) as a dummy statement (6.0.1.c) in those rare situations where the use of a completer is inappropriate, e.g. l: <u>skip</u>) 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. <u>go to</u> exit in y := <u>if</u> x ≥ 0 <u>then</u> sqrt(x) <u>else</u> <u>go to</u> exit <u>fi</u>, or f in (j > a | f | j) from 10.2.3.r.

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 {19 July 1968}

{Coercends are voided when it is required that their values (and therefore ~~its~~ 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,h} :
   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 elaboration (i.e. preelaborating 1.1.6.f) a voided-coercend is discarded.}

{In the range 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 {19 July 1968}

8.3.0.1. Syntax

a) MODE confrontation{81a,820d,e,f,g,821a,b,822a,823a,b,824a,
    825a,b,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 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. <u>loc</u> <u>real</u> := 3.14, which is contained in <u>real</u> 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 ; <u>loc</u> <u>real</u> := 3.14 ;   d) <u>loc</u> <u>real</u> ;
e) x ;   f) 1 ; 3.14 }

8.3.1.2. Semantics {24 July 1968}

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

c) 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} and if the given name is not nil, then Step 2 is taken; {otherwise, the further elaboration is undefined;}
Step 2: 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 is a multiple value or a structured value, then Step 3 is taken; otherwise, the considered value is superseded {a} by a new instance of the given value 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 {6.3.2.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 new 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) does not begin with 'reference to' but with 'row of' or 'structured with'.}

8.3.2. Conformity relations {22 July 1968}

{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.}

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 an becomes symbol{31c}.

8.3.2.2. Semantics

A conformity-relation is elaborated in the following steps:

Step 1: Its tertiaries are elaborated collaterally {6.3.2.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.

{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 ; rb := 1 ; rb := 1.0, each of the three assignments 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 a range with the declarations struct cons = (strons car, strons cdr) ; union strons = (ref string, ref cons) ; cons cell := (string := "abc", nil), the identity-relation cdr of cell :=: nil has the value true, because nil possesses a unique name, though it refers to no value.}

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} ;
  cosoft 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{6.3.2.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 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   {22 July 1968}

{Formulas are either dyadic, e.g. x + i, or monadic, e.g. abs x. A formula has at least one operator or value-of-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 value-of-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,h,i,j}.

b) MOID PRIORITY formula{81b,820d,e,f,g,821a,b,822a,b,c,823a,b,
  824a,825a,b,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{d} :
  firm MODE monadic operand{f}.

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

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

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

{Examples:

b). x + y   d) x × y ; x ;

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

h) val xx }

8.4.2. Semantics {22 July 1968}

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 its operator at its operator-defining occurrence {7.5.2, 4.3.2.b} ;

Step 2: The copy {which is now a closed-clause} 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 new instance 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-prelude (10.2.0).

priority									monadic
1	2	3	4	5	6	7	8	9	
minus	∨	∧	=	<	-	×	↑		¬ - + **down**
plus			≠	≤	+	÷			**abs** **bin** **repr**
times				≥		÷:			**leng** **short**
over				>		/			**odd** **sign**
modb									**round** **entier**
prus									**re** **im** **conj** **up**

Observe that 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 value-of-symbol in a depression is gentle rather than firm because its elaboration should not involve dereferencing.}

8.5. Cohesions {22 July 1968}

{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 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

{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, call 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 may be actual-parameters (see 7.4.1.b) and therefore may appear in declarations. }

8.5.1.1. Syntax {22 July 1968}

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 non-
local 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 range, which is the scope of the
name possessed by x (2.2.4.2.c). The clo$\overset{s}{\wedge}$ed-clause
((ref real xx ; real x := pi ; xx := x) = pi)
however, possesses the value true. }

8.5.2. Selections {19 July 1968}

{A selection selects a field from a structured value, e.g. re of z selects the first real field (usually called 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 a range containing the declarations:

```
  struct language = (int age, ref language father) ;
  language algol := (10, (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,g,h and the remarks below 7.1.1 and 8.5.1.2). The use of an identifier which is the same sequence of symbols as a field-selector in the same range 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-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.d,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-named-[age]-and-[language]-named-[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 {24 July 1968}

{Bases are denotations, e.g. 3.14, identifiers, e.g. x, slices, e.g. x1[i] and clause-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,822a,b,c,823a,b,824a,825a, 826a,828b} : MOID slice{861a,-} ; MOID clause call{862a} ; MOID denotation{510b,511a,512a,513a,514a,52a,53a,54b,55a,-}; MOID identifier{41b,-}.

{Examples:
a) x1[i,j] ; set random(x) ; 3.14 ; x }

8.6.0.2. Semantics

An identifier is elaborated by considering a new instance of the value, if any, possessed by its defining occurrence {4.1. 2, 7.4.2.Step 5} ; its value is then the considered value.

8.6.1. Slices

{Slices are obtained by subscripting, e.g. x1[i] or by trimming, e.g. x1[2:n], or by a mixture of both, e.g. x2[j:n,j] or x2[,k]. Subscripting and trimming may be done only to primaries, e.g. x1 and x2 or (p | x1 | y1). The value of a slice may be either one element of the value of its primary, e.g. x1[i] is a real value from the row of real x1, or a subset of the elements, e.g. x2[i] is the i-th row of the matrix x2.}

8.6.1.1. Syntax

a) REFETY ROWSETY ROWWSETY NONROW slice{860a} :
weak REFETY ROWS ROWWSETY NONROW primary{811d},
sub symbol{31e}, ROWS leaving ROWSETY indexer{b,c},
bus symbol{31e}.

```
b)  row of ROWS leaving row of ROWSETY indexer{a,b} :
      trimmer{f} option, comma symbol{31e},
                                ROWS leaving ROWSETY indexer{b,c,d,e} ;
      subscript{k}, comma symbol{31e},
                                ROWS leaving row of ROWSETY indexer{b,d}.
c)  row of ROWS leaving EMPTY indexer{a,b,c} :
      subscript{k}, comma symbol{31e},
                                ROWS leaving EMPTY indexer{c}.
d)  row of leaving row of indexer{b} : trimmer{f} option.
e)  row of leaving EMPTY indexer{b} : subscript{k}.
f)  trimmer{b,d} : actual lower bound{71s}, up to symbol{31e},
      actual upper bound{71s}, new lower bound part{g} option,
      new increment part{i} option.
g)  new lower bound part{f} : at symbol{31e}, new lower bound{h}.
h)  new lower bound{g} : strong integral unit{61e}.
i)  new increment part{f} : by symbol{31e}, new increment{j}.
j)  new increment{i} : strong integral unit{61e}.
k)  subscript{b,c,e} : strong integral unit{61e}.
l)* trimscript : trimmer{f} option, subscript{k}.
m)* indexer : ROWS leaving ROWSETY indexer{b,c,d,e}.
```

```
    {Examples:
a)  x1[i] ; x2[i,j] ; x2[i] ; x1[2:n] ;
b)  2:m,j ; 1,2:n ;
c)  i,j ;
d)  2:n ;
e)  i ;
f)  2:n ; 2:n at 0 ; 2:m at 0 by 2 ;
g)  at 0 ;
h)  0 ;
i)  by 2 ;
j)  2 ;
k)  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.s, 2:3<u>at</u>0 ; 2:n ; 2:<u>by</u> 2 ; 2: ; :5 and :<u>at</u>0 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, new-lower-bounds and new-increments of its indexer are elaborated collaterally {6.3.2.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 $d_i > 0$ and $l_i \leq k \leq u_i$ or $d_i < 0$ and $u_i \leq k \leq l_i$, 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 ;

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

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

if now either $b > 0$, $l_i \leq l$ and $u \leq u_i$ or $b < 0$, $u_i \geq l$ and $u \geq l_i$, then the offset in the copy is increased by $(l - l_i) \times d_i$, and then l_i is replaced by l', u_i by $l' + (u + b - l) \div b - 1$ and d_i by $d_i \times b$; 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 that 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, if the value of the new-increment is positive (negative), then the value of the subscript is restricted to run from the value of the strict-lower-(upper-) bound to the value of the strict-upper-(lower-)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 (highest) 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 and the assignations

```
  y1 := x1[1:n by -1] ; x1 := y1
```

reverse the order of the elements of x1.}

8.6.2. Clause calls {24 July 1968}

8.6.2.1. Syntax

a) MOID clause call{860a} :
    firm procedure with PARAMETERS MOID primary{811d},
                    actual PARAMETERS{54e,74b} pack.

{Examples:
a) sin(x) }

8.6.2.2. Semantics

A clause-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 clause-call is replaced by that copy ;

Step 3: That copy {which is now a closed-clause} 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 clause-call taken in the same order ;

Step 5: The elaboration of the copy is initiated; its value, if any, is that of the clause-call; if this elaboration is completed or terminated, then the copy is replaced by the clause-call before the elaboration of a successor is initiated.

{The clause-call samelson(m, (int j)real : (x1[j])) as contained in the range 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 unitary-clauses {Chapter 8},
D 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,
E for a serial-clause {6.1.1.a},
F for a unitary-clause,
G for two or more unitary-clauses separated by comma-symbols,
H for a declarer {7.1},
I, J, K and L for identifiers {4.1},
<u>L</u> for zero or more long-symbols,
M for an identifier,
N for an indication {4.2},
O for zero or one identifiers,
P for a virtual-plan {5.4.1.c},
Q for a choice-clause {6.4.1.c},
R for a routine-denotation {5.4},
S for a statement {6.0.1.c},
T for a unitary-clause,
U for zero or one virtual-declarers{7.1.1.b},
V for a virtual-declarer,
W for a unitary-clause, and
Z for a formal-declarer {7.1.1.b} all of whose formal-row-of-rowers {7.1.1.q} are empty.

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** ZI = **loc** H where Z and H specify the same mode {7.1.2.a} may be replaced by HI.

{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** N = **struct** may be replaced by **struct** N = and **mode** N = **union** may be replaced by **union** N =.

{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 unitary-declaration (formal-parameter {5.4.1.f}, field-declarator {7.1.1.g}) and another unitary-declaration (formal-parameter, field declarator) following a comma-symbol following the given unitary-declaration (formal-parameter, field-declarator) both begin with an occurrence of the mode-symbol, of the structure-symbol, of the union-of-symbol, of the priority-symbol, of the operation-symbol, of one same actual-declarer, or of one same formal-declarer, 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) A void-symbol which follows a procedure-symbol or a parameters-pack {7.1.1.z} may be omitted.

{e.g., **proc** **void** p := stop may be replaced by **proc** p :=

stop and **proc**(**real**)**void** q := set random may be replaced by **proc**(**real**) q := set random but the void-symbol in **void** : stop may not be omitted.}

e) If each corresponding pair of constituent declarers in P and R specifies the same mode, then

  proc PI = R may be written **proc** I = R,

  op PN = R may be written **op** N = R, and

  proc PO:=R may be written **proc** O := R.

  {e.g., **proc**(**ref** **int**) incr = (**ref** **int**) : (i := i + 1) may be replaced by **proc** incr = (**ref** **int** i) : (i := i + 1), **op**(**ref** **int**)**int** **decr** = (**ref** **int** i)**int** : (i := i - 1) may be replaced by **op** **decr** = (**ref** **int** i)**int** : (i := i - 1) and **proc**(**real**)**int** p := (**real** x)**int** : (**round** x), obtained from **ref** **proc**(**real**)**int** p = **loc** **proc**(**real**)**int** := (**real** x)**int** : (**round** x) by 9.2.a, may be replaced by **proc** p := (**real** x)**int** : (**round** x).}

9.3. Repetitive statements

a) The statement {6.0.1.c}

```
  begin(int J := F, int K = B, L = T) ;
    M: if D(K > 0 | J ≤ L |: K < 0 | J ≥ L | true) then
    int I = J ; (W | S ; (DJ := J + K) ; go to M)
    fi
  end   ,
```

where J, K, L and M do not occur in D, W or S, and where I differs from J and K, may be replaced by

```
  for I from F by B to T while W do S   ,
```

and if, moreover, I does not occur in W or S, then **for** I **from** may be replaced by **from**.

b) The statement

```
  begin(int J := F, int K = B) ;
   M: (int I = J ; (W | S ; (DJ := J + K) ; go to M))
  end    ,
```

9.3. continued {25 July 1968}

where J, K and M do not occur in D, W or S, and where I differs from J and K, may be replaced by
  for I **from** F **by** B **while** W **do** S ,
and if, moreover, I does not occur in W or S, then **for** I **from** may be replaced by **from**.

c) **from** 1 **by** may be replaced by **by**.

d) **by** 1 **to** may be replaced by **to**, and **by** 1 **while** may be replaced by **while**.

e) **while** **true** **do** may be replaced by **do**.

{e.g., **for** i **from** 1 **by** 1 **to** n **while** **true** **do** x := x + a may be replaced by **to** n **do** x := x + 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** Q **fi** **fi** may be replaced by **elsf** Q **fi** and
then **if** Q **fi** **fi** may be replaced by **thef** Q **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** DI = 1 **then** B **elsf** D(I = 2 | **true**) **then** C **fi**), where I does not occur in B, C or D, may be replaced by **case** A **in** B, C **esac** {or by (A| B, C)}.

c) (**int** I = A ; **if** DI = 1 **then** B **else** **case**(DI = 1) **in** G **esac** **fi**), where I does not occur in C, D or G, may be replaced by

case A **in** C, G **esac** {or by (A | C, G)}.

{Examples of the use of such "case" clauses are given in 11.11.w, ap.}

9.5. Complex values

(D **L** **compl** I = (A, B) ; I)
where I does not occur in D, may be replaced by (A ⊥ B) {or by (A **i** B)}.

10. Standard declarations

a) A "standard declaration" is one of the constituent declarations of the standard-prelude {2.1.b} {; it is either an "environment enquiry" supplying information concerning a specific property of the implementation (2.3.c), a "standard priority" or "standard operation", a "standard mathematical constant or function", a "synchronization operation" or a "transput declaration"} .

b) A representation of the standard-prelude is obtained by altering each form in 10.1, 10.2, 10.3, 10.4 and 10.5 in the following steps:

Step 1: Each sequence of symbols between ⦃ and ⦄ in a given form is altered in the following steps:

Step 1.1: If ***D*** occurs in the given sequence of symbols, then the given sequence is replaced by a chain of a sufficient number of sequences separated by comma-symbols; the first new sequence is a copy of the given sequence in which copy ***D*** is deleted; the n-th new sequence, n > 1, is a copy of the given sequence in which copy ***D*** is replaced by a sub-symbol followed by n-2 comma-symbols followed by a bus-symbol ;

Step 1.2: If, in the given sequence of symbols, as possibly modified in Step 1.1, ***L int*** (***L real*** or ***L compl***) occurs, then that sequence is replaced by a chain of *int lengths* {10.1.a} (*real lengths* {10.1.c}) sequences separated by comma-symbols, the n-th new sequence being a copy of the given sequence in which copy each occurrence of *L*(***L***) has been replaced by (n-1) times *long*(***long***) ;

Step 2: Each occurrence of ⦃ and ⦄ in a given form, as possibly modified in Step 1, is deleted ;

Step 3: If, in a given form, as possibly modified in Steps 1 and 2, ***L int*** (***L real*** or ***L compl***, ***L bits*** or ***L abs***, both ***L int*** and ***L real*** or both ***L int*** and ***L compl***) occurs, then the form is replaced by a sequence of *int lengths* {10.1.a} (*real lengths* {10.1.c}, *bits widths* {10.1.f}, the minimum of *int lengths* and *real lengths*) new forms; the n-th new form is a copy of the given form in which copy each occurrence of *L*(***L***, ***K***, ***S***) is replaced by (n-1) times *long*(***long***, ***leng***, ***short***) ;

Step 4: If ***P*** occurs in a given form, as possibly modified or made in the Steps above, then the form is replaced by four new forms obtained by replacing ***P*** consistently throughout the form by either - or + or × or / ;

Step 5: If *Q* occurs in a given form, as possibly modified or made in the Steps above, then the form is replaced by four new forms obtained by replacing *Q* consistently throughout the form by either *minus* or *plus* or *times* or *over* ;

Step 6: If *R* occurs in a given form, as possibly modified or made in the Steps above, then the form is replaced by six new forms obtained by replacing *R* consistently throughout the form by either < or ≤ or = or ≠ or ≥ or > ;

Step 7: Each occurrence of *F* in any form, as possibly modified or made in the Steps above, is replaced by a representation of the letter-aleph-symbol {1.2.1.n, 5.5.1.6};

Step 8: If, in some form, as possibly modified or made in the Steps above, % occurs followed by the representation of an identifier (field-selector, indication), then that occurrence of % is deleted and each occurrence of the representation of that identifier (field-selector, indication) in any form is replaced by the representation of one same identifier (field-selector, indication) which does not occur elsewhere in a form, and Step 8 is taken ;

Step 9: If a representation of a comment occurs in any form, as possibly modified or made in the Steps above, then this representation is replaced by a representation of an actual-declarer or closed-clause suggested by the comment ;

Step 10: If, in any form, as possibly modified or made in the Steps above, a representation of a routine-denotation occurs whose elaboration involves the manipulation of real numbers, then this denotation may be replaced by any other denotation whose elaboration has approximately the same effect {The degree of approximation is left undefined in this Report (see also 2.2.3.1.c).} ;

Step 11: The standard-prelude is that declaration-prelude whose representation is the same as the sequence of all the forms, as possibly modified or made in the Steps above.

{The declarations in this Chapter are intended to describe their effect clearly. The effect may very well be obtained by a more efficient method. }

10.1. Environment enquiries

a) **int** int lengths = **c** the number of different lengths of integers **c** ;

b) **L** **int** L max int = **c** the largest L integral value **c** ;

c) **int** real lengths =
   c the number of different lengths of real numbers **c** ;

d) **L** **real** L max real = **c** the largest L real value **c** ;

e) **L** **real** L small real = **c** the smallest L real value such that both
   L1 + L small real > **L1** and **L1** - L small real < **L1** **c** ;

f) **int** bits widths =
   c the number of different widths of standard bit rows **c** ;

g) **int** L bits width =
   c the number of bits in a standard L bit row; see **L** **bits** {10.2.6.a} **c** ;

h) **op** **abs** = (**char** a) **int** :
   c the integral equivalent of the character 'a' **c** ;

i) **op** **repr** = (**int** a) **char** :
   c that character 'x', if it exists, for which **abs** x = a **c** ;

10.2. Standard priorities and operations

10.2.0. Standard priorities

a) **priority** **minus** = 1, **plus** = 1, **times** = 1, **over** = 1, **modb** = 1, **prus** = 1,
   ∨ = 2, ∧ = 3, = = 4, ≠ = 4, < = 5, ≤ = 5, ≥ = 5, > = 5, - = 6,
   + = 6, × = 7, ÷ = 7, ÷: = 7, / = 7, ↑ = 8 ;

10.2.1. Operations on boolean operands

a) **op** ∨ = (**bool** a, b) **bool** : (a | **true** | b) ;

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

c) **op** ¬ = (**bool** a) **bool** : (a | **false** | **true**) ;

d) **op** = = (**bool** a, b) **bool** : ((a ∧ b) ∨ (¬a ∧¬ b)) ;

e) **op** ≠ = (**bool** a, b) **bool** : (¬(a = b)) ;

f) **op** **abs** = (**bool** a) **int** : (a | 1 | 0) ;

10.2.2. Operations on integral operands

a) **op** < = (**L** **int** a, b) **bool** : **c** true if the value of 'a' is smaller than
   that of 'b' and false otherwise **c** ;

b) **op** ≤ = (**L** **int** a, b) **bool** : (¬(b < a)) ;

c) **op** = = (**L** **int** a, b) **bool** : (a ≤ b ∧ b ≤ a) ;

d) **op** ≠ = (**L** **int** a, b) **bool** : (¬(a = b)) ;

10.2.2. continued

```
e) op ≥ = (L int a, b) bool : (b ≤ a) ;
f) op > = (L int a, b) bool : (b < a) ;
g) op - = (L int a, b) L int :
     c the value of 'a' minus that of 'b' c ;
h) op - = (L int a) L int : (L0 - a) ;
i) op + = (L int a, b) L int : (a - - b) ;
j) op + = (L int a) L int : (a) ;
k) op abs = (L int a) L int : (a < L0 | -a | a) ;
l) op × = (L int a, b) L int : (L int s := L0, i := abs b ;
     while i ≥ L1 do(s := s + a ; i := i - L1) ; (b < L0 | -s | s)) ;
m) op ÷ = (L int a, b) L int :
     (b ≠ L0 | L int q := L0, r := abs a ;
     while(r := r - abs b) ≥ L0 do q := q + L1 ;
     (a < L0 ∧ b ≥ L0 ∨ a ≥ L0 ∧ b < L0 | -q | q)) ;
n) op ÷: = (L int a, b) L int : (a - a ÷ b × b) ;
o) op / = (L int a, b) L real : (L real c = a, d = b; c / d) ;
p) op ↑ = (L int a, int b) L int :
     (b ≥ 0 | L int p := L1; to b do p := p × a; p) ;
q) op leng = (L int a) long L int : c the long L integral value equivalent
     to the value of 'a' c ;
r) op short = (long L int a) L int : c the L integral value, if it exists,
     equivalent to the value of 'a' c ;
s) op odd = (L int a) bool : (abs a ÷: L2 = L1) ;
t) op sign = (L int a) int : (a > L0 | 1 |: a < L0 | -1 | 0) ;
```

10.2.3. Operations on real operands

```
a) op < = (L real a, b) bool : c true if the value of 'a' is
     smaller than that of 'b' and false otherwise c ;
b) op ≤ = (L real a, b) bool : (¬(b < a)) ;
c) op = = (L real a, b) bool : (a ≤ b ∧ b ≤ a) ;
d) op ≠ = (L real a, b) bool : (¬(a = b)) ;
e) op ≥ = (L real a, b) bool : (b ≤ a) ;
f) op > = (L real a, b) bool : (b < a) ;
g) op - = (L real a, b) L real : c the value of 'a' minus that of 'b' c
     {2.2.3.1.c}
h) op - = (L real a) L real : (L0 - a) ;
```

10.2.3. continued

```
i) op + = (L real a, b) L real : (a - - b) ;
j) op + = (L real a) L real : (a) ;
k) op abs = (L real a) L real : (a < L0 | -a | a) ;
l) op × = (L real a, b) L real : c the value of 'a' times that of 'b' c ;
      {2.2.3.1.c}
m) op / = (L real a, b) L real : c the value of 'a' divided by that of
      'b' c ; {2.2.3.1.c}
n) op leng = (L real a) long L real :
      c the long L real value equivalent to the value of 'a' c ;
o) op short = (long L real a) L real : c the L real value, if it
      exists, equivalent to the value of 'a' c ;
p) op round = (L real a) L int : c a L integral value, if one exists,
      equivalent to a L real value differing by not more than one-half
      from the value of 'a' c ;
q) op sign = (L real a) int : (a > L0 | 1 |: a < L0 | -1 | 0) ;
r) op entier = (L real a) L int : (L int j := L0 ;
      (j ≤ a | e : j := j + L1 ; (j ≤ a | e | j - L1) |
               f : j := j - L1 ; (j > a | f | j))) ;
```

10.2.4. Operations on arithmetic operands

```
a) op P = (L real a, L int b) L real : (L real c = b; a P c) ;
b) op P = (L int a, L real b) L real : (L real c = a; c P b) ;
c) op R = (L real a, L int b) bool : (L real c = b; a R c) ;
d) op R = (L int a, L real b) bool : (L real c = a; c R b);
e) op ↑ = (L real a, int b) L real : (L real p := L1 ;
      to abs b do p := p × a ; (b ≥ 0 | p | L1 / p)) ;
```

10.2.5. Complex structures and associated operations

```
a) struct L compl = (L real re, im) ;
b) op re = (L compl a) L real : (re of a) ;
c) op im = (L compl a) L real : (im of a) ;
d) op abs = (L compl a) L real :(L sqrt(re a ↑ 2 + im a ↑ 2));
e) op conj = (L compl a) L compl : (re a ⊥ - im a) ;
f) op = = (L compl a, b) bool : (re a = re b ∧ im a = im b) ;
g) op ≠ = (L compl a, b) bool : (¬(a = b)) ;
h) op + = (L compl a) L compl : (a) ;
```

10.2.5. continued

```
i) op - = (L compl a) L compl : (- re a ⊥ - im a) ;
j) op + = (L compl a, b) L compl : (re a + re b ⊥ im a + im b) ;
k) op - = (L compl a, b) L compl : (re a - re b ⊥ im a - im b) ;
l) op × = (L compl a, b) L compl :
      (re a × re b - im a × im b ⊥ re a × im b + im a × re b) ;
m) op / = (L compl a, b) L compl :
      (L real d = re(b × conj b) ; L compl n = a × conj b ;
      (re n / d ⊥ im n / d)) ;
n) op leng = (L compl a) long L compl : (leng re a ⊥ leng im a) ;
o) op short = (long L compl a) L compl : (short re a ⊥ short im a) ;
p) op P = (L compl a, L int b) L compl : (L compl c = b; a P c) ;
q) op P = (L compl a, L real b) L compl : (L compl c = b; a P c) ;
r) op P = (L int a, L compl b) L compl : (L compl c = a; c P b) ;
s) op P = (L real a, L compl b) L compl : (L compl c = a; c P b) ;
t) op ↑ = (L compl a, int b) L compl : (L compl p := L1 ;
     to abs b do p := p × a ; (b ≥ 0 | p | L1 / p)) ;
```

10.2.6. Bit rows and associated operations

```
a) mode L bits = [1 : L bits width] bool ; {10.1.g}
b) op = = ([1 : int n] bool a, b) bool :
     (for i to n do(a[i] ≠ b[i] | l) ; true. l : false) ;
c) op ≠ = ([] bool a, b) bool : (¬(a = b)) ;
d) op ∨ = ([1 : int n] bool a, b) [] bool : ([1 : n] bool c ;
     for i to n do c[i] := a[i] ∨ b[i] ; c) ;
e) op ∧ = ([1 : int n] bool a, b) [] bool : ([1 : n] bool c ;
     for i to n do c[i] := a[i] ∧ b[i] ; c) ;
f) op ≤ = ([] bool a, b) bool : (a ∨ b = b) ;
g) op ≥ = ([] bool a, b) bool : (b ≤ a) ;
h) op ↑ = ([1 : int n] bool a, int b) [] bool : ([1 : n] bool c := a ;
     to abs b do (b > 0 | for i from 2 to n do
       c[i - 1] := c[i] ; c[n] := false
       | for i from n by - 1 to 2 do
       c[i] := c[i - 1] ; c[1] := false); c) ;
i) op L abs = (L bits a) L int : (L int c := L0 ;
     for i to L bits width do c := L2 × c + abs a[i] ; c) ;
j) op bin = (L int a) L bits : if a ≥ L0 then L int b := a; L bits c ;
     for i from L bits width by - 1 to 1 do
     (c[i] := odd b; b := b ÷ L2); c fi ;
```

10.2.7. Operations on character operands

```
a) op < = (char a, b) bool : (abs a < abs b) ; {10.1.h}
b) op ≤ = (char a, b) bool : (¬ (b < a)) ;
c) op = = (char a, b) bool : (a ≤ b ∧ b ≤ a) ;
d) op ≠ = (char a, b) bool : (¬ (a = b)) ;
e) op ≥ = (char a, b) bool : (b ≤ a) ;
f) op > = (char a, b) bool : (b < a) ;
```

10.2.8. String mode and associated operations

```
a) mode string = [1: ] char ;
b) op < = ([1 : int m] char a, [1 : int n] char b) bool :
     (int i := 1; int p = (m < n | m | n); bool c ;
     (p < 1 | n ≥ 1 | e : (c := a[i] = b[i] |: (i := i + 1) ≤ p | e) ;
                            (c | m < n | a[i] < b[i]))) ;
c) op ≤ = (string a, b) bool : (¬ (b < a)) ;
d) op = = (string a, b) bool : (a ≤ b ∧ b ≤ a) ;
e) op ≠ = (string a, b) bool : (¬ (a = b)) ;
f) op ≥ = (string a, b) bool : (b ≤ a) ;
g) op > = (string a, b) bool : (b < a) ;
h) op R = ([1 : int n] char a, char b) bool : (string c = b; a R c) ;
i) op R = (char a, [1 : int n] char b) bool : (string c = a; c R b) ;
j) op + = ([1 : int m] char a, [1 : int n] char b) string :
     ([1 : m + n] char c ;
     c[1 : m] := a ; c[m + 1 : m + n] := b ; c) ;
k) op + = (string a, char b) string : (string s = b ; a + s) ;
l) op + = (char a, string b) string : (string s = a ; s + b) ;
```

{The operations defined in b, h and i imply that if *abs* "a" < *abs* "b", then "" < "a" ; "a" < "b" ; "aa" < "ab" ; "aa" < "ba" ; "ab" < "b". }

10.2.9. Operations combined with assignations

```
a) op minus = (ref L int a, L int b) ref L int : (a := a - b) ;
b) op minus = (ref L real a, L real b) ref L real : (a := a - b) ;
c) op minus = (ref L compl a, L compl b) ref L compl : (a := a - b) ;
d) op plus = (ref L int a, L int b) ref L int : (a := a + b) ;
e) op plus = (ref L real a, L real b) ref L real : (a := a + b) ;
f) op plus = (ref L compl a, L compl b) ref L compl : (a := a + b);
g) op times = (ref L int a, L int b) ref L int : (a := a × b) ;
```

10.2.9. continued

h) **op** **times** = (**ref** **L** **real** a, **L** **real** b) **ref** **L** **real** : (a := a × b) ;
i) **op** **times** = (**ref** **L** **compl** a, **L** **compl** b) **ref** **L** **compl** : (a := a × b) ;
j) **op** **over** = (**ref** **L** **int** a, **L** **int** b) **ref** **L** **int** : (a := a ÷ b) ;
k) **op** **modb** = (**ref** **L** **int** a, **L** **int** b) **ref** **L** **int** : (a := a ÷: b) ;
l) **op** **over** = (**ref** **L** **real** a, **L** **real** b) **ref** **L** **real** : (a := a / b) ;
m) **op** **over** = (**ref** **L** **compl** a, **L** **compl** b) **ref** **L** **compl** : (a := a / b) ;
n) **op** **Q** = (**ref** **L** **real** a, **L** **int** b) **ref** **L** **real** : (a **Q**(**L** **real** := b)) ;
o) **op** **Q** = (**ref** **L** **compl** a, **L** **int** b) **ref** **L** **compl** : (a **Q**(**L** **compl** := b)) ;
p) **op** **Q** = (**ref** **L** **compl** a, **L** **real** b) **ref** **L** **compl** : (a **Q**(**L** **compl** := b)) ;
q) **op** **plus** = (**ref** **string** a, **string** b) **ref** **string** : (a := a + b) ;
r) **op** **prus** = (**ref** **string** a, **string** b) **ref** **string** : (a := b + a) ;
s) **op** **plus** = (**ref** **string** a, **char** b) **ref** **string** : (a := a + b) ;
t) **op** **prus** = (**ref** **string** a, **char** b) **ref** **string** : (a := b + a) ;

10.3. Standard mathematical constants and functions

a) **L** **real** L pi = **c** a L real value close to π ; see Math. of Comp. v. 16, 1962, pp. 80-99 **c** ;
b) **proc** L sqrt = (**L** **real** x) **L** **real** : **c** if x ≥ 0, a L real value close to the square root of 'x' **c** ;
c) **proc** L exp = (**L** **real** x) **L** **real** : **c** a L real value, if one exists, close to the exponential function of 'x' **c** ;
d) **proc** L ln = (**L** **real** x) **L** **real** : **c** a L real value, if one exists, close to the natural logarithm of 'x' **c** ;
e) **proc** L cos = (**L** **real** x) **L** **real** : **c** a L real value close to the cosine of 'x' **c** ;
f) **proc** L arccos = (**L** **real** x) **L** **real** : **c** if **abs** x ≤ **L1**, a L real value close to the inverse cosine of 'x', **L0** ≤ L arccos(x) ≤ L pi **c** ;
g) **proc** L sin = (**L** **real** x) **L** **real** : **c** a L real value close to the sine of 'x' **c** ;
h) **proc** L arcsin = (**L** **real** x) **L** **real** : **c** if **abs** x ≤ **L1**, a L real value close to the inverse sine of 'x', **abs** L arcsin(x) ≤ L pi / **L2** **c** ;
i) **proc** L tan = (**L** **real** x) **L** **real** :
c a L real value, if one exists, close to the tangent of 'x' **c** ;
j) **proc** L arctan = (**L** **real** x) **L** **real** :
c a L real value close to the inverse tangent of 'x', **abs** L arctan(x) ≤ L pi / **L2** **c** ;

10.3. continued

k) ***proc*** *L random =* ***L real expr c*** *the next pseudo-random L real value from a uniformly distributed sequence on the interval [****L0****,* ***L1****)* ***c*** *;*

l) ***proc*** *L set random = (****L real*** *x) : (****c*** *the next call of L random is made to deliver the value of 'x'* ***c*** *; L random) ;*

10.4. Synchronization operations

a) ***op down*** *= (****ref int*** *dijkstra) : (****do elem****(****if*** *dijkstra ≥ 1* ***then*** *dijkstra* ***minus*** *1 ; l* ***else c*** *if the closed-statement replacing this comment is contained in a unitary-phrase which is a constituent unitary-phrase of the smallest collateral-phrase, if any, beginning with a parallel-symbol and containing this closed-statement, then the elaboration of that unitary-phrase is halted* {6.0.2.c} *; otherwise, the further elaboration is undefined* ***c fi****); l :* ***skip****) ;*

b) ***op up*** *= (****ref int*** *dijkstra) :* ***elem****(dijkstra* ***plus*** *1 ;* ***c*** *the elaboration is resumed of all phrases whose elaboration is not terminated but is halted because the name possessed by 'dijkstra' referred to a value smaller than one* ***c****) ;*

{For insight into the use of ***down*** and ***up***, see E.W. Dijkstra, Cooperating Sequential Processes, EWD123, Tech. Univ. Eindhoven, 1965. }

10.5. Transput declarations {"So it does!" said Pooh. "It goes in!"
"So it does!" said Piglet. "And it comes out!"
"Doesn't it?"said Eeyore. "It goes in
and out like anything."
Winnie-the-Pooh, A.A. Milne.}

10.5.0. Transput modes and straightening

10.5.0.1. Transput modes

a) *mode % simplout = union(ǂ L int ǂ, ǂ L real ǂ, ǂ L compl ǂ,*
bool, char, string) ;

b) *mode % outtype = union(ǂ D L int ǂ, ǂ D L real ǂ, ǂ D bool ǂ,*
ǂ D char ǂ, ǂ D outstruct ǂ) ;

c) *mode % outstruct = c an actual-declarer specifying a mode united from* {2.2.4.1.h} *all modes, except that specified by tamrof, which are structured from* {2.2.4.1.j} only modes from which the mode specified by outtype is united *c*;

d) *mode % intype = union(ǂ ref D L int ǂ, ǂ ref D L real ǂ,*
ǂ ref D bool ǂ, ǂ ref D char ǂ, ǂ ref D outstruct ǂ) ;

e) *mode % tamrof = struct(string F) ;* {See the remarks under 5.5.1.6.}

10.5.0.2. Straightening

a) *op % straightout = (outtype x) [] simplout :*
c the result of "straightening" 'x' c ;

b) *op % straightin = (intype x) [] ref simplout :*
c the result of straightening 'x' c ;

The result of straightening a given value is obtained in the following steps:

Step 1: If the given value is (refers to) a value from whose mode that specified by *simplout* is united, then the result is a new instance of a multiple value composed of a descriptor consisting of an offset 1 and one quintuple (1, 1, 1, 1, 1) and the (the name of the) given given value as its only element, and Step 4 is taken;

Step 2: If the given value is (refers to) a multiple value, then, letting n stand for the number of elements of that value, and y_i for the result of straightening its i-th element, Step 3 is taken; otherwise, letting n stand for the number of fields of the (of the value referred to by the) given value, and y_i for the result of straightening its i-th field, Step 3 is taken;

Step 3: If the given value is not (is) a name, then, letting m_i stand for the number of elements of y_i, the result is a new instance of a multiple value composed of a descriptor consisting of an offset 1 and one quintuple $(1, m_1 + \ldots + m_n, 1, 1, 1)$ and elements, the l-th of which, where $l = m_1 + \ldots + m_{k-1} + j$, is the (is the name referring to the) j-th element of y_k for $k = 1, \ldots, n$ and $j = 1, \ldots, m_k$.

Step 4: If the given value is not (is) a name, then the mode of the result is 'row of' ('row of reference to') followed by the mode specified by *simplout*.

10.5.1. Channels and files

aa) {"Channels", "backfiles" and files model the transput devices of the physical machine used in the implementation.

bb) A channel corresponds to a device, e.g. a card reader or punch, a magnetic drum or disc, to part of a device, e.g. a piece of core memory, the keyboard of a teleprinter, or to a number of devices, e.g. a bank of tape units or even a set-up in nuclear physics the results of which are collected by the computer. A channel has certain properties (10.5.1.1.d: 10.5.1.1.n).
The transput devices of some physical machine may be seen in more than one way as channels with properties. The choice made in an implementation is a matter for individual taste. Some possible choices are given in table I.

cc) All information on a given channel is to be found in a number of backfiles. A backfile (10.5.1.1.b) comprises a threedimensional array of integers (bytes of information), the *book* of the backfile, indexed by *page*, *line* and *char*; the lower bounds of the *book* are all one, the upperbounds are nonnegative integers, the *maxpage*, *maxline* and *maxchar* of the backfile; furthermore, the backfile comprizes the position of the "end of file", i.e. the page number, line number and char number up to which the backfile is filled with information, and the "identification-string" of the backfile.

dd) After the elaboration of the declaration of *chainbfile* (10.5.1.1.c) backfiles form the chains of backfiles referenced by *chainbfile* , each backfile chained to the next one by its field *next*.

Examples

i) In a certain implementation, channel six is a line printer. It has no input information, *chainbfile* [*6*] is initialized to refer to a backfile the *book* of which is an integer array with upper bounds 2000, 60 and 144 (2000 pages of continuous stationery), with the end of file at position (1, 1, 1), and *next* equal to nil. All elements of the *book* are left undefined.

ii) Channel four is a drum, divided into 32 segments each being one page of 256 lines of 256 bytes. It has 32 backfiles of input information (the previous contents of the drum), so *chainbfile* [*4*]

properties	card reader	card punch	magnetic tape unit			line printer
reset possible	false	false	true	true	true	false
set possible	false	false	false	false	false	false
get possible	true	false	true	true	false	false
put possible	false	true	false	true	true	true
bin possible	false	true	false	true	false	false
max page	1	1	very large	very large	very large	very large
max line	large	very large	16	large	60	60
max char	72	80	84	large	144	144
stand conv	a 48- or 64-character code		64-char code	some code	line-pr code	line-pr code
max nmb files	1	1	1	1	1	1

properties	magnetic disc	magnetic drum		paper tape reader		tape punch
reset possible	true	true	true	false	false	false
set possible	true	false	true	false	false	false
get possible	true	true	true	true	true	false
put possible	true	true	true	false	false	true
bin possible	true	true	true	false	true	false
max page	200	1	1	1	1	1
max line	16	1	256	very large	very large	very large
max char	128	524288	256	80	150	4
stand conv	some code	some code	some code	5-hole code	7-hole code	lathe code
max nmb files	10	4	32	1	1	1

TABLE I: Properties of some possible channels

is initialized to refer to the first backfile of a chain of 32 backfiles, the last one having *next* equal to nil. Each of those backfiles has an end of file at position (2, 1, 1).

iii) Channel twenty is a tape unit, it can accommodate one tape at a time, one input tape is mounted, and another tape laid in readiness. Here *chainbfile* [*20*] is initialized to refer to a chain of two backfiles.

Since it is part of the standard declarations, all input is part of the program, though not of the particular program.

ee) A file (10.5.1.2.a) is a structure which comprizes a reference to a backfile, and the information necessary for the transput routines to work with that backfile. A backfile is associated with a file by means of *open* (10.5.1.2.b) or *create*(10.5.1.2.e). With a given channel a certain maximum number (10.5.1.1.m) of files may be associated at any stage of the elaboration. The association is ended by means of *scratch* (10.5.1.2.u), *close* (10.5.1.2.s) or *lock* (10.5.1.2.t).

ff) When a file is "opened" on a channel for which *idf possible* is false, then the first backfile is taken from the chain of bfiles for that channel, and is made the *bfile* of the file, obliterating the previous backfile, if any, of the file.
When a file is opened on a channel for which *idf possible* is true, then the backfile is taken from the chain of backfiles for the channel, which has the given identification string; this backfile is made the *bfile* of the file.

gg) When a file is "created" on a channel, then a backfile is generated (8.5.) with a *book* of the maximum size for the channel, the end of file at (1,1,1) and an empty string as its identification string.

hh) When a file is "scratched" its association with a channel ends and its associated backfile is obliterated.

ii) When a file is "closed" its association with a channel ends and its backfile is attached to the chain referenced by *chainbfile* of that channel. The identification of the backfile is made to be the given identification string.

jj) When a file is "locked" its association with a channel ends and its backfile is attached to the chain referenced by *lockedbfile* for that channel. The identification of the bfile is made to be the given identification string.

kk) A file comprizes procedures, which may be provided by the programmer, which are called when in transput certain error-conditions occur. They are:

a) *logical file ended*, which is called when during input from a file on a channel, for which set possible is false, the end of file of its backfile is transgressed;

b) *physical file ended*, which is called when the *maxpage*, the *maxline* or the *maxchar* of the backfile of a file is transgressed.

c) *disagreement*, which is called when during formatted transput a character is transput which does not agree with the frame specifying it (for instance when under control of *f̲ d f̲* a space is read) or when the conversion fails (see ll);

d) *incompatibility*, which is called when during formatted transput an attempt is made to transput a value under control of a picture with which it is incompatible.

ll) The *conv* of a file is used in conversion; if *conv* of the file is nil, then *stand conv* of the channel is used as "conversion key", and, otherwise, the string to which *conv* refers, which may be provided by the programmer.

On output, if the character to be converted is not the same as some element of the conversion key, then *disagreement* of the file is called; otherwise, the character is converted to an integer, viz. the lowest among the ordinal numbers of those elements of the key which are the same as that character.

On input, if an integer to be converted is larger than the number of elements of the conversion key, then *disagreement* of the file is called; otherwise, the integer is converted to that character in the key whose ordinal number is that integer.

mm) The *term* of a file is used in reading strings of a variable number of characters, where either the end of line or any of the characters of *term* serves as a terminator (see 5.5.1.nn and 10.5.2.2.dd). This terminatorstring may be provided by the programmer.

nn) On a channel for which *reset possible* is true, a file may be "reset", causing its position to be (1, 1, 1). Resetting while writing a file on a channel for which *reset possible* is false first sets the end of file at the current position.

oo) On a channel for which *set possible* is true, a file may be "set", causing its position to be the given position.

pp) On files opened on a channel for which *set possible* is false, binary and nonbinary transput may not be alternated, i.e. after opening, creating or resetting such a file, either is possible, but, once one has taken place on the file, the other may not until the file has been reset again.

qq) On files opened on a channel for which *set possible* is false, and *put possible* and *get possible* both true, input and output may be alternated, but before, after some output, any input is done, the end of file is first set at the current position.

rr) When in transput something happens which is left undefined, for instance by an explicit call of *undefined* (10.5.1.2), this does not imply that the elaboration is catastrophically and immediately terminated, but only that the action which takes place is not or cannot be described by this Report alone, and is generally implementation dependent. For instance, in some implementation it may be possible to set a tape unit to any position within the logical file, even though set possible is false.

Example:

```
begin  file f1, f2; [1 : 10000] int x; int n;
       open (f1, "my input", channel 2);
       f2 := f1; c now f1 and f2 can be used interchangeably c
       val conv of f1 := flexocode; c flexocode is a string,
       defined in the library declarations for this implementation;
       f1 and f2 both use flexocode c
       conv of f2 := string := telexcode;
       c now f1 and f2 use different codes c
       reset (f1); c consequently f2 is reset too c
       for i while ¬logical file ended (f1) do
       (n := i; get (f1, x [i]));
       c too bad if there are more than 10000 integers in the input c
       reset (f1);
       for i to n do put (f2, x [i]);
       reset (f2); close (f2, "my output");
       c f1 is now closed too c
end}
```

10.5.1.1. Channels

```
a) int nmb channels = c an integral-clause indicating the number of
      transput devices in the implementation c;
b) struct % bfile = ([,,]int book, int lpage, lline, lchar,
      maxpage, maxline, maxchar, string idf, ref bfile next);
c) [1 : nmb channels] ref bfile % chainbfile := c some appropriate
      initialization (see 10.5.1.dd) c
d) [1 : nmb channels] bool reset possible = c a row-of-boolean-clause,
      indicating which of the physical devices corresponding to the
      channels allow resetting {e.g. rewinding of a magnetic tape} c ;
e) [1 : nmb channels] bool set possible = c a row-of-boolean-clause,
      indicating which devices can be accessed at random c ;
f) [1 : nmb channels] bool get possible = c a row-of-boolean-clause,
      indicating which devices can be used for input c ;
g) [1 : nmb channels] bool put possible = c a row-of-boolean-clause,
      indicating which devices can be used for output c ;
h) [1 : nmb channels] bool bin possible = c a row-of-boolean-clause,
      indicating which devices can be used for binary transput c ;
i) [1 : nmb channels] bool idf possible = c a row-of-boolean-clause,
      indicating on which devices backfiles have an identification c;
j) [1 : nmb channels] int max page = c a row-of-integral-clause,
      giving the maximum number of pages per file for the channels c;
k) [1 : nmb channels] int max line = c a row-of-integral-clause,
      giving the maximum number of lines per page c;
l) [1 : nmb channels]  int max char = c a row-of-integral-clause,
      giving the maximum number of characters per line c;
m) [1 : nmb channels] ref string % stand conv = c a row-of-reference-to-
      row-of-character-clause giving the standard conversion keys for
      the channels c;
n) [1 : nmb channels] int max nmb files = c a row-of-integral-clause,
      giving the maximum numbers of files the channels can accomodate c;
o) [1 : nmb channels] int % nmb opened files;
      for i to nmb channels do nmb opened files [i]:= 0;
p) [1 : nmb channels] ref bfile % lockedbfile;
      for i to nmb channels do lockedbfile [i]:= nil
q) proc file available = (int channel) bool:
      (nmb opened files [channel] < max nmb files [channel]);
```

10.5.1.2. Files

```
a) struct file = (ref bfile % bfile,
                  ref int % page, % line, % char, % char, % forp,
                  ref bool % state def, % state get, % state bin, % opened,
                  ref string % format, conv, term,
                  ref proc logical file ended, physical file ended,
                           disagreement, incompatibility);
b) proc open = (ref bfile file, string idf, int ch):
   if file available (ch)
   then bfile bf := chainbfile[ch];
        while bf :≠: nil do
            (idf of bf = idf ∨¬idf possible [ch]|l|bf := next of bf);
        undefined. l:
        file := (bfile := bf, int := 1, int := 1, int := 1, int := ch, int := 0,
        bool := false, bool, bool,   bool := true, nil, nil, string := "",
        proc := skip, proc := skip, proc := skip, proc := skip);
        chainbfile[ch] := next of chainbfile[ch];
        nmb opened files[ch] plus 1
   else undefined
   fi;
c) proc create = (ref file file, int ch):
   if file available (ch)
   then bfile bf = ([1 : maxpage[ch], 1 : maxline[ch], 1 : maxchar[ch]]int,
        1, 1, 1, maxpage[ch], maxline[ch], maxchar[ch],"", nil);
        file := (bfile := bf, int := 1, int := 1, int := 1, int := ch, int := 0,
        bool := false, bool, bool,   bool := true, nil, nil, string := "",
        proc := skip, proc := skip, proc := skip, proc := skip);
        nmb opened files [ch] plus 1
   else undefined
   fi;
d) proc set = (file file, int p, l, c):
        if set possible[chan of file] ∧ opened of file
        then page of file := p; line of file := l; char of file := c;
            check plc(file)
        else undefined
        fi;
```

```
e) proc reset = (file file) :
     if reset possible[chan of file] ∧ opened of file
     then if state def of file ∧ ¬ state get of file ∧
     ¬ set possible[chan of file]
          then lpage of bfile of file := page of file ;
               lline of bfile of file := line of file ;
               lchar of bfile of file := char of file
          fi ;
          page of file := line of file := char of file := 1 ;
          state def of file := false
     else undefined
     fi ;
f) proc % check plc = (file file) bool: (opened of file |:
     logical file ended (file) | logical file ended of file |:
     outside (file) | physical file ended of file);
g) proc % outside = (file file) bool:
     (line ended (file) ∨ page ended (file) ∨ file ended (file)).
h) proc line ended = (file file) bool:
    (opened of file | char of file > maxchar of bfile of file);
i) proc page ended = (file file) bool:
     (opened of file | line of file > maxline of bfile of file);
j) proc logical file ended = (file file) bool: (opened of file|:
       set possible [char of file] ∧ state def of file ∧ state get of file |
    int p = page of file, lp = lpage of bfile of file,
        l = line of file, ll = lline of bfile of file,
        c = char of file, lc = lchar of bfile of file;
    (p < lp | false |: p > lp | true |: l < ll | false |:  l > ll | true |
        c ≥ lc) | false);
k) proc file ended = (file file) bool:
     (opened of file | page of file > max page of bfile of file)
```

```
l) proc % get string = (file file, ref[1 : int n] char s):
     if get possible[chan of file] ∧ opened of file
     then [1 : int m] char conv = (conv of file :=: nil | stand conv [chan of file]|
          conv of file);
   ref int p = page of file, l = line of file, c = char of file;
     if ¬ set possible[chan of file] thef state def of file
     then (state bin of file | undefined) fi;
     state def of file := state get of file := true;
     state bin of file := false
     for i to n do (check plc(file); book of bfile of file [p, l, c] > m |
        disagreement of file; s[i] := "."; c plus 1 |
     s[i] := conv [book of bfile of file [p, l, c]]; c plus 1)
   else undefined
   fi;

m) proc % put string = (file file,[1 : int n] char s) :
     if put possible[chan of file] ∧ opened of file
     then int ch = chan of file, p = page of file, l = line of file ;
          string conv = (conv of file :=: nil | stand conv[ch] |
                conv of file) ; int space, h ; ref int c = char of file ;
          if ¬ set possible[ch] thef state def of file
          then (state bin of file | undefined);
               (state get of file |
               lpage of bfile of file := p;
               lline of bfile of file := l; lchar of bfile of file := c;
               state get of file := false)
          else state def of file := true ; state get of file :=
               state bin of file := false ;
               (¬ char in string(".", space, conv) | undefined) ;
               for i to max page[ch] do for j to max line[ch] do
               for k to max char[ch] do
                    book of bfile of file[i, j, k] := space
          fi ;
          for i to n do (check plc (file);¬ char in string (s [i], h , conv) |
             disagreement of file | book of bfile of file[p, l, c]:=h; c plus 1)
     else undefined
     fi;
```

```
n) proc char in string = (char c, ref int i, [1 : int w bool b] char s) bool:
      (for k to w do(c = s[k] | i := k ; l) ; false. l : true) ;
o) proc space = (file file) :
      (char of file plus 1 ; check plc (file));
p) proc backspace = (file file) :
      (char of file minus 1 ; check plc (file));
q) proc new line = (file file) :
      (line of file plus 1 ; char of file := 1 ; check plc (file));
r) proc new page = (file file):
      (page of file plus 1; line of file := char of file := 1; check plc(file));
s) proc close = (file file, string idf):
      (opened of file | int ch = chan of file; idf of bfile of file := idf;
      next of bfile of file := chainbfile[ch];
        chainbfile[ch] := bfile := bfile of file;
      opened of file := false; nmb opened files[ch] minus 1);
t) proc lock = (file file, string idf):
      (opened of file | int ch = chan of file; idf of bfile of file := idf;
      next of bfile of file := lockedbfile[ch];
        lockedbfile[ch] := bfile := bfile of file;
      opened of file := false; nmb opened files[ch] minus 1);
u) proc scratch = (file file):
      (opened of file | opened of file := false;
      nmb opened files[chan of file] minus 1);
v) proc char number = (file f) int : (opened of f | char of f);
w) proc line number = (file f) int : (opened of f | line of f);
x) proc page number = (file f) int : (opened of f | page of f);
```

10.5.1.3. Standard channels and files

a) ***int** stand in channel* = ***c** an integral-clause such that get possible [stand in channel] is true, idf possible [stand in channel] is false and stand conv [stand in channel] comprizes, in some order, all character-tokens c;*

b) ***int** stand out channel* = ***c** an integral-clause such that put possible [stand out channel] is true, idf possible [stand out channel] is false and stand conv [stand out channel] comprizes, in some order, all character-tokens **c**;*

c) ***int** stand back channel* = ***c** an integral-clause such that reset possible [stand back channel], set possible [stand back channel], get possible [stand back channel], put possible [stand back channel] and bin possible [stand back channel] are true, idf possible [stand back channel] is false and stand conv [stand back channel] comprizes, in some order, all character-tokens **c**;*

d) ***file** % f; open (f, "", stand in channel);*
   ***file** stand in = f;*

e) *open (f, "", stand out channel);*
   ***file** stand out = f;*

f) *open (f, "", stand back channel);*
   ***file** stand back = f;*

{Certain "standard files" (d, e, f) need not (and cannot) be opened by the programmer, but are opened for him in the standard declarations; *print* (10.5.2.1.a) can be used for output on *stand out*, *read* (10.5.2.2.a) for input from *stand in*, and *write bin* (10.5.4.1.a) and *read bin* (10.5.4.2.a) for transput involving *stand back*.}

10.5.2. Formatless transput

10.5.2.1. Formatless output

{For formatless output, *print* and *put* can be used. The elements of the given value of the mode specified by [] *union (outtype, proc (file))* are treated one after the other; if an element is of the mode specified by *proc (file)* (i.e. a "layout procedure"), then it is called with the file as its parameter; otherwise, it is straightened (10.5.0.2), and the resulting values are output on the given file one after the other, as follows:

aa) If the mode of the value is specified by *L int*, then first, if there is not enough room on the line for *L int width + 2* characters, then this room is made by giving a new line and, if the page is full, giving a new page; then the value is output as if under control of the picture *"."n(L int width - 1)z+d.*

bb) If the mode of the value is specified by *L real*, then, first, if there is not enough room on the line for *L real width + L expwidth + 5* characters, then this room is made; then the value is output as if under control of the picture *"."+d.n(L real width - 1)den(L expwidth - 1)z+d.*

cc) If the mode of the value is specified by *L compl*, then first the real part is output as in bb; then the string possessed by *". ⊥"* is output as in dd; finally, the imaginary part is output.

dd) If the mode of the value is specified by [] *char* then its elements are written one after the other.

ee) If the mode of the value is specified by *char* then, first if the line is full room is made; then the character is written.

ff) If the mode of the value is specified by *bool* then, if the value is true (false) a flip (flop) is output as in ee.}

```
a) proc print = ([] union (outtype, proc(file))x):
      put (stand out, x);
```

10.5.2.1. continued

```
b) proc put = (file file, [1 : int n] union (outtype, proc(file))x):
   begin outtype ot; proc(file)pf;
         for i to n do
         (ot ::= x[i]; pf ::= x[i] | pf(file)|
         [1 : int l] simplout y = straightout ot;
         for j to l do
         (string s ; bool b ; char c ;
         (ǂ (L int i ; (i ::= y[j] |
            s := L int string(i, L int width + 1, 10) ;
            sign supp zero(s, 1, L int width  4 ))) ǂ) ;
         (ǂ (L real x ; (x ::=y[j] | s := L real string
            (x, L real width + L exp width + 4, L real width - 1, L exp width) ;
            sign supp zero(s, L real width + 5, L real width + L exp width + 4))) ǂ) ;
         (ǂ (L compl z ; (z ::= y[j] |
            put(file, (re z, "⊥⊥", im z)) ; end)) ǂ) ;
         (b ::= y[j] | s := (b | "1" | "0")) ;
         (c ::= y[j] | nextplc(file) ; put string(file, c) ; end) ;
         (s ::= y[j] | putstring(file, s); end);
         [1 : int n] char t := "." + s; int c1 = char of file;
         char of file := c1 + n; (outside (file) | nextplc(file)|
         char of file := c1); put string (file, t);
         end: skip))
    end;
```

10.5.2.l.continued 2

```
c) proc L int string = (L int x, int w, r) string : (r > 1 ∧ r < 17 |
      string c :=""; L int n := abs x; L int lr = Kr ;
      for i to w - 1 do(c prus dig char(S(n ÷: lr)) ; n over lr) ;
      (n = L0 | (x ≥ L0 | "+" | "-") + c)) ;
 d) proc L real string = (L real x, int w, d, e) string :
      (d ≥ 0 ∧ e > 0 ∧ d + e + 4 ≤ w |
       L real g = L10 ↑ (w - d - e - 4) ; L real h = g × L.1 ;
       L real y := abs x; int p := 0;
       while y ≥ g do(y times L.1 ; p plus 1) ;
       (y > L0 | while y < h do(y times L10 ; p minus 1)) ;
       (y + L.5 × L.1 ↑ d ≥ g | y := h ; p plus 1) ;
       L dec string((x ≥ 0 | y | -y), w - e - 2, d) +
         "10" + int string(p, e + 1, 10)) ;
 e) proc L dec string = (L real x, int w, d) string :
      (abs x < L10 ↑ (w - d - 2) ∧ d ≥ 0 ∧ d + 2 ≤ w | string s :="";
        L real y := (abs x + L.5 × L.1 ↑ d) × L.1 ↑ (w - d - 2);
        for i to w - 2 do s plus dig char((int c = entier S(y times L10) ;
                y minus Kc ; c)) ;
          (x ≥ 0 | "+" | "-") + s[1 : w - d - 2] + "." + s[w - d - 1 : ]);
 f) proc % dig char = (int x) char : ("0123456789abcdef"[x + 1]) ;
```

{In connection with 10.5.2.c, d, e, see Table II. }

```
 g) proc % sign supp zero = (ref string c, int l, u) :
     for i from l + 1 to u while c[i] = "0" do
      (c[i] := c[i - 1] ; c[i - 1] := "."[1]) ;
 h) int L int width = (int c := 1;
      while L10 ↑ (c - 1) < L.1 × L max int do c plus 1 ; c) ;
 i) int L real width = - entier S(L ln(L small real) / L ln(L10)) ;
 j) int L exp width = 1 + entier S
        (Lln(Lln(L max real) / Lln(L10)) / Lln(L10));
k) proc % nextplc = (file file) : (opened of file |
      (line ended(file) | new line(file)) ;
      (page ended(file) | new page(file)) ;
      (logical file ended(file) | logical file ended of file | ;
      file ended(file) | physical file ended of file));
```

L int string : $$\underbrace{+\overbrace{\mathrm{DDDDDDDDDDD}}^{w-1}}_{w}$$

L dec string : $$\underbrace{+\overbrace{\mathrm{DDDDDD}}^{w-d-2}.\overbrace{\mathrm{DDDDDDDD}}^{d}}_{w}$$

L real string : $$\underbrace{+\overbrace{\mathrm{DDDDDDDD}}^{w-d-e-4}.\overbrace{\mathrm{DDDDDD}}^{d}{}_{10}\pm\overbrace{\mathrm{DDD}}^{e}}_{w}$$

TABLE II: Display of the values of
L int string, *L dec string* and *L real string*

frame

[*1*]	type (1 = integer, 2 = real fixed, 3 = real floating, 4 = complex fixed, 5 = complex floating, 6 = string, 7 = integer choice, 8 = boolean)
[*2*]	radix (2, 4, 8, 10 or 16)
[*3*]	sign (0 = no sign frame, 1 = sign frame '+', 2 = sign frame '-')
[*4*]	number of digits before point; if type = 1 then *w-1*, else if type = 2 or 4 then *w-d-2* else if type = 3 or 5 then *w-d-e-4*, or, if type = 6, then number of characters in string
[*5*]	number of digits after point; if type = 2, 3, 4 or 5 then *d*
[*6*]	sign of exponent; if type = 3 or 5 then as [*3*]
[*7*]	number of digits of exponent; if type = 3 or 5 then *e*
[*8*], ..., [*14*]	as [*1*], ..., [*7*] when *frame*[*1*] = 4 or 5

TABLE III: Significance of the elements of *frame*

10.5.2.2. Formatless input

{For formatless input, *read* and *get* can be used. The elements of the given value of the mode specified by [] *union (intype, proc (file))* are treated one after the other; if an element is a layout procedure, then it is called with the file as its parameter; otherwise, it is straightened (10.5.0.2), and to the resulting names values are assigned, input from the given file as follows:

aa) If the name refers to a value whose mode is specified by *L int*, then, first the file is searched for the first character that is not a space (giving new lines and pages as necessary); then the largest string is transscribed from the file that could be input under control of some picture of the form *n(k2)d* or *+n(k1)* *"."n(k2)d;* this string is converted to an integer by *L string int*.

bb) If the name refers to a value whose mode is specified by *L real*, then, first the file is searched for the first character that is not a space; then the largest string is transscribed from the file that could be input under control of a picture of the form *+n(k1)"."n(k2)d* or *n(k2)d* followed by *.n(k3)d* or *s.* possible followed by *n(k4)"."en(k5)"."+n(k6)"."n(k7)d* or *n(k4)"."en(k6)"."n(k7)d*; this string is converted to a real number by *L string real*.

cc) If the name refers to a value whose mode is specified by *L compl*, then, first a real number is assigned to the real part, input as in bb; then the file is searched for the first character that is not a space; then a plus-i-times is required; finally, a real number is input and assigned to the imaginary part.

dd) If the name refers to a value whose mode is specified by [] *char*, then if both upper- and lowerstate of the value are one then as many characters are read as the value has elements; otherwise, characters are read from the line under control of the terminatorstring referenced by the file (5.5.1.nn, 10.5.1.mm); the string with those characters as its elements is then the resulting value.

ee) If the name refers to a value whose mode is specified by *char* then, first, if the line is full a new line is given, and, if the page is full, a new page is given; then the character is read from the file.

ff) If the name refers to a value whose mode is specified by *bool* then, first the file is searched for the first character that is not a space; then a character is read; if this character is flip (flop), then the resulting value is true (false); if the character is neither flip nor flop, then the further elaboration is undefined.}

```
a) proc read = ([] union (intype, proc(file))x):
       get (stand in, x);
b) proc get = (file file, [1 : int w] union (intype, proc(file))x):
   begin intype it; proc(file)pf; char k;
        for i to w do
        (it ::= x[i]; pf ::= x[i] | pf(file)|
        [1 : int l] ref simplout y = straightin it;
        op ? = (string s) bool :
        (outside(file) | false |: get string(file, k) ;
          char in string(k, loc int, s) |
          true | backspace(file) ; false) ;
        proc skip spaces = void: (while (nextplc(file); ? ".") do skip);
        proc read dig = string :
             (string t :=""; while ? "0123456789" do t plus k; t);
        proc read num = string :
             (char t := (?"+-" | k | "+"); while ? "." do skip;
             (char in string (k, loc int, ".100123436789" | t + read dig |
             disagreement of file; "0"));
        proc read real = string :
             (string t  := (skip spaces; read num); (?"." | t plus "." + read dig);
             (? "10" | t plus "10" + read num); t)
        for j to l do
        (ref bool bb; ref char cc; ref string ss;
        (ⱡ (ref L int ii ; (ii  ::= y[j] |
          val ii  := L string int(read num, 10))) ⱡ) ;
        (ⱡ (ref L real xx ; (xx  ::= y[j] |
          val xx  := L string real(read real))) ⱡ) ;
        (ⱡ (ref L compl zz ; (zz  ::= y[j] | get(file, re of zz) ;
          (skip spaces ; ? "⊥" | get(file, im of zz) | undefined))) ⱡ) ;
       (bb ::= y[j] | skip spaces ; val bb := (? "10" | k = "1")) ;
```

```
    (cc ::= y[j] | nextplc(file); get string(file, cc));
    (ss ::= y[j] | ref [bool bl, bool bu] char tt = ss;
      (bl ∧ bu | get string(file, ss)|
      string t := ""; while ¬ (line ended(file) ∨ ? term of file) do t plus k;
      val ss := t));
      end: skip end ;
c) proc L string int = ([1 : int w] char x, int r) L int : (r > 1 ∧ r < 17|
      L int n := L0; L int lr = Kr ; for i from 2 to w do
      n := n × lr + K(int d = char dig(x[i]) ; (d < r | d)) ;
      (x[1] = "+" | n |: x[1] = "-" | -n)) ;

d) proc L string real = (string x) L real :
      (int e ; (char in string("10", e, x) |
      L string dec(x[1 : e - 1]) × L10 ↑ string int(x[e + 1 :], 10)|
        L string dec(x)) ;
e) proc L string dec = ([1 : int w] char x) L real :
      (L real r := L0; int p; (char in string(".", p, x) |
        [1 : w - 2] char s = x[2 : p - 1] + x [p + 1:];
        for i to w - 2 do r := L w × r +
          K(int d = char dig(x[i]) ; (d < 10 | d)) ;
        (x[1] = "+" | r |: x[1] = "-" | -r) × L.1 ↑ (w - p) |
          L string dec(x + ".")) ;
f) proc % char dig = (char x) int :
      (int i ; (char in string(x, i, "0123456789abcdef") | i - 1)) ;
```

10.5.3. Formatted transput

{For the significance of formats see format-denotations (5.5).}

```
a) proc format = (file file, tamrof tamrof):
     (forp of file := 1; format of file := collection list pack
      ("(" + F of tamrof + ")", loc int := 1));
b) proc % collection list pack = (string s, ref int p) string:
      (string t := collection(s, p);
      while s[p] = "," do t plus "," + collection(s, p); p plus 1; t);
c) proc % collection = (string s, ref int p) string:
      (int n, q; string f := (p plus 1; insertion(s, p));
      q := p; replicator(s, p, n);
      (s[p] = "(" | string t = collection list pack(s, p);
      to n do f plus t | p := q; f plus picture(s, p, loc[1 : 14] int));
       f + insertion(s, p)) ;
 d) proc % insertion = (string s, ref int p) string :
       (int q = p ; skip insertion(s, p) ; s[q : p - 1]);
 e) proc % skip insertion = ([1 : int l false] char s, ref int p):
      while(p > l | false |: skip align(s, p) | true |
       skip lit(s, p)) do skip ;
 f) proc % skip align = (string s, ref int p) bool :
       (int q = p ; replicator(s, p, loc int) ;
       (char in string(s[p], loc int, "x y p l k") |
        p plus 1 ; true | p := q ; false)) ;
 g) proc % replicator = (string s, ref int p, n) :
       (string t := ""; while char in string
        (s[p], loc int, "0123456789") do (t plus s[p] ; p plus 1);
        n := (t = "" | 1 | string int("+" + t, 10))) ;

h) proc % skip lit = (string s, ref int p) bool :
      (int q = p ; replicator(s, p, loc int) ;
       (s[p] = """" | while(s[p plus 1] = """" | s[p plus 1] = """" |
        true) do skip ; true | p := q ; false)) ;
```

```
i) proc % picture = ([1 : int m.false] char format, ref int p,
          ref[] int frame) string :
     begin int n ; int po = p;
          op ? = (string s) bool :
          (skip insertion(format, p) ; p > m | false |
           int q = p ; replicator(format, p, n) ;
            (format[p] = "s" | p plus 1) ;
             (char in string(format [p], loc int, s)|
                p plus 1; true | p := q; false));
          proc intreal pattern = (ref[1 : 7] int frame) bool :
          ((num mould(frame[2 : 4    ]) | frame[1] := 1 ; l) ;
           (? "." |: num mould(frame[3 : 5    ]) | frame[1] := 2 ; l) ;
           (? "e" |: num mould(frame[5 : 7    ]) | frame[1] := 3 ; l) ;
           false. l : true) ;
          proc num mould = (ref[1 : 3] int frame) bool :
          ((? "r" | frame[1] := n) ; (? "z" | frame[3] plus n) ;
           (? "+" | frame[2] := 1 |: ? "-" | frame[2] := 2) ;
           while ? "dz" do frame[3] plus n ;
           format[p] = "," v format[p] = "i" v format[p]= ")");
          proc string mould = (ref[] int frame) bool : (?"t" | true |
           while ? "a" do frame[4] plus n ; format[p] = "," v format[p]= ")");
          for i to 14 do frame[i] := 0 ; frame[z] := 10;
          (intreal pattern(frame[1 : 7]) | (? "i" |
           frame[1] plus 2 ; intreal pattern(frame[8 : 14    ])) ; end) ;
          (string mould(frame) | frame[1] := 6 ; end) ;
          (? "b" | frame[1] := 8 | p plus 1 ; frame[1] := 7) ;
          (format[p] = "(" |
           while ? "(," do skip lit(format, p) ; p plus 1) ;
     end:.skip insertion(format, p); format [po : p - 1]
    end ;
```

{In connection with 10.5.3.i see table III.}

10.5.3.1. Formatted output

```
a) proc outf = (file file, tamrof tamrof, [] outtype x) :
      (format(file, tamrof) ; out(file, x)) ;
b) proc out = (file file, [1 : int n] outtype x) :
   begin string format = format of file ; ref int p = forp of file ;
         for k to n do
         ([1 : int l] simplout y = straightout x[k] ;
         for j to l do
         ([1 : 14] int frame ; int q := p; picture(format, p, frame) ;
         (frame[1] | int, real, real, compl, compl, string, intch, bool) ;
     int: (⌈ (L int i ; (i ::= y[j] |
         trans edit L int(file, i, format, q, frame) ; end)) ⌉); incomp;

    real: (⌈ (L real x ; (x ::= y[j] |
         trans edit L real(file, x, format, q, frame) ; end)) ⌉);
         (⌈ (L int i ; (i ::= y[j] |
         trans edit L real(file, i, format, q, frame) ; end)) ⌉); incomp;

   compl: (⌈ (L compl z ; (z ::= y[j] |
         trans edit L compl(file, z, format, q, frame) ; end)) ⌉) ;
         (⌈ (L real x ; (x ::= y[j] |
         trans edit L compl(file, x, format, q, frame) ; end)) ⌉) ;
         (⌈ (L int i ; (i ::= y[j] |
         trans edit L compl(file, i, format, q, frame) ; end)) ⌉);  incomp;

string: ([1 : frame[4]] char s ; char c ;
        (s ::= y[j] |: frame[4] = 0 | put(file, s) |
        trans edit string(file, s, format, q, frame) ; end) ;
        (c ::= y[j] |
        trans edit string(file, c, format, q, frame) ; end)); incomp;

 intch: (int i ; (i ::= y[j] |
        trans edit choice(file, i, format, q) ; end)); incomp;
  bool: (bool b ; (b ::= y[j] |
        trans edit bool(file, b, format, q) ; end)) ;
incomp: incompatibility of file; put(file, y[j]).
   end: do insertion(file, format, q); p plus 1))
end ;
```

10.5.3.1. continued

```
c) proc % trans edit L int = (file f, L int i, string format,
     ref int p, [] int fr) :
     trans edit string(f, L int string(i, fr[4] + 1, fr[2]), format, p, fr) ;
d) proc % trans edit L real = (file f, L real x, string format,
     ref int p, [] int fr) :
     trans edit string(f, stringed L real(x, fr), format, p, fr) ;
e) proc % stringed L real = (L real x, [] int fr) string :
     (fr[1] = 2 | L dec string(x, fr[4] + fr[5] + 2, fr[5]) |
     L real string(x, fr[4] + fr[5] + fr[7] + 4, fr[5], fr[7])) ;
f) proc % trans edit L compl = (file f, L compl z, [] int fr) :
     trans edit string(f, ([1 : 14] int g := fr; g 1  minus 2;
                  stringed L real(re z, g[1 : 7]) + "⊥" + stringed L real
                  (im z, g[8 : 14    ])), format, p, fr) ;
g) proc % trans edit string = (file f, string x, [1 : int m  false] char format,
     ref int p, [] int frame) :
     begin int p1 := 1, n; bool supp; string s := x;
           op ? = (string s) bool :
           (do insertion(file, format, p) ; p > m | false |
            int q = p ; replicator(format, p, n) ;
            (supp := format[p] = "s" | p plus 1) ;
            (char in string(format [p], loc int, s)|
               p plus 1; true | p := q; false));
           proc copy = void: ((  supp | put string(f, s[p1])); p1 plus 1);
           proc intreal mould = void:
           (? "r" ; sign mould(frame[3]) ; int mould ;
            (? "." | copy ; int mould |: s[p1] = "." | p1 plus 1) ;
            (? "e" | copy ; sign mould(frame[6]) ; int mould)) ;
           proc sign mould = (int sign) : (sign = 0 | p1 plus 1 |
            s[p1] := (s[p1] = "+" | (sign | "+", ".") | "-") ;
           (? "z" | sign supp zero (s, p1, p1 + n)| n := 0);
            put string(file, s[p1 : p1 + n : 1]) ; p plus 1; p1 plus n);
           proc int mould = void:
           (l : (? "z" | bool zs := true; to n do
                 (s[p1] = "0" ∧ zs | put string(file, ".") ;
                 p1 plus 1 | zs := false ; copy) ; l) ;
                (? "d" | to n do copy ; l)) ;
```

```
          proc string mould = expr while ? "a" do to n do copy ;
      tes: (frame[1] = 6 | string mould |: intreal mould ;
           frame[1] > 3 | p plus 1 ; copy ; intreal mould)
    end ;
h) proc % trans edit choice = (file f, int c, string format, ref int p) :
    (c > 0 | do insertion(f, format, p) ; p plus 2 ;
    to c - 1 do(skip lit(format, p) ; format[p] = "," |
     p plus 1 | undefined) ;
    do lit(f, format, p) ;
    while format[p] ≠ ")" do(p plus 1 ; skip lit(format, p)) ;
     p plus 1 | undefined) ;
i) proc % trans edit bool = (file f, bool b, string format, ref int p) :
    (do insertion(f, format, p) ; (format[p + 1] = "(" |
     p plus 2 ; (b | do lit(f, format, p) ; p plus 1 ; skip lit
     (format, p) | skip lit(format, p) ; p plus 1 ; do lit(f, format, p)) |
     put string(f, (b | "1" | "0"))) ; p plus 1) ;
j) proc % do insertion = (file f, [1 : int l false] char s, ref int p):
    while(p > l | false |: do align(f, s, p) | true |
     do lit(f, s, p)) do skip ;
k) proc % do align = (file f, string s, ref int p) bool :
    (int q = p ; int n ; replicator(s, p, n) ;
     (s[p] = "x" | to n do space(f) ; l |:
    s[p] = "y" | to n do backspace(f) ; l |:
    s[p] = "p" | to n do new page(f) ; l |:
    s[p] = "l" | to n do new line(f) ; l |:
    s[p] = "k" | char of f := n ; l) ; p := q ; false.
     l : p plus 1 ; true) ;
l) proc % do lit = (file f, string s, ref int p) bool :
    (int q = p ; int n ; replicator(s, p, n) ; (s[p] = """" |
    while(s[p plus 1] = """" | s[p plus 1] = """" | true) do
    put string(f, s[p]) ; true | p := q ; false)) ;
```

10.5.3.2. Formatted input

```
a) proc inf = (file file, tamrof tamrof, [] intype x) :
       (format(file, tamrof) ; in(file, x)) ;
b) proc in = (file file, [1 : int n] intype x) :
   begin string format = format of file ; ref int p = forp of file ;
               for k to n do
               ([1 : int l]ref simplout y = straightin x[k] ;
               for j to l do
               ([1 : 14] int frame ; int q := p; picture(format, p, frame) ;
               (frame[1] | int, real, real, compl, compl, string, intch, bool) ;
         int: (∤ (ref L int ii ; (ii ::= y[j] |
               trans indit L int(file, ii, format, q, frame) ; end)) ∤) ; incomp;
        real: (∤ (ref L real xx; (xx ::=  y[j] |
              trans indit L real(file, xx, format, q, frame); end)) ∤); incomp;
       compl: (∤ (ref L compl zz; (zz ::= y[j] |
              trans indit L compl(file, zz, format, q, frame); end)) ∤); incomp;
      string: (ref string ss ; ref char cc ; [1 : frame[4]] char t ;
              (frame[4] = 0 |: ss ::= y[j] | get(file, ss) ; end | undefined) ;
              trans indit string(file, t, format, q, frame) ;
              (ss ::= y[j] | val ss := t ; end |: cc ::= y[j] |
              val cc := t[1] ; end)) ; incomp;
       intch: (ref int ii; (ii ::= y[j] |
              trans indit choice(file, ii, format, q); end)); incomp;
        bool: (ref bool bb; (bb ::= y[j] |
              trans indit bool(file, bb, format, q); end));
      incomp: incompatibility of file; undefined.
         end: req insertion(file, format, q); p plus 1))
     end;

c) proc % trans indit L int =
      (file f, ref L int i, string format, ref int p, [] int fr) :
      (string t ; trans indit string(f, t, format, p, fr) ;
       i := L string int(t, fr[2]));
d) proc % trans indit L real =
      (file f, ref L real x, string format, ref int p, [] int fr) :
      (string t ; trans indit string(f, t, format, p, fr) ;
       x := L string real(t)) ;
```

```
e) proc % trans indit L compl =
     (file f, ref L compl z, string format, ref int p, [] int fr) :
     (string t ; int i ; trans indit string(f, t, format, p, fr) ;
      z := (char in string("⊥", i, t) |
            (L string real(t[1 : i - 1]) ⊥ L string real(t[i + 1 : /]))));
f) proc % trans indit string =
     (file f, ref string t, [1 : int m false] char format,
      ref int p, [] int frame) :
    begin int n ; bool supp ; char k ; string x := "";
          op ? = (string s) bool :
          (req insertion(format, p) ; p > m | false |
           int q = p ; replicator(format, p, n) ;
           (supp := format[p] = "s" | p plus 1) ;
           (char in string(format [p], loc int, s)|
              p plus 1; true | p := q; false));

          priority ! = 8;
          op ! = (string s, string c) string :
           (char in string (k, loc int, s) | (supp | "" | k) |
            disagreement of file; c);
          proc next = char: (get string (f, k); k);
          proc intreal mould = void :
          (?"r"; sign mould (frame[3]); int mould;
           (?"." | x plus "." ! "."; int mould);
           (?"e" | x plus "10" ! "10"; sign mould (frame[6]); int mould));
          proc sign mould = (int sign): (sign = 0 | x plus "+" |
          int j; ( ¬ ? "z" | n := 0); for i to n + 1 while next = "." do j := i;
           x plus (sign = 1 | "+-" ! "+" | : k = "-" | k | "+");
           (char in string (k, loc int, "0123456789") | x plus k);
           for i from j + 1 to n + 1 do x plus (next; "0123456789"!"0"));
          proc int mould = void: (l:
           (?"z" | int j; for i to n while next = "." do j := i;
           x plus "0123456789" ! "0";
           for i from j + 1 to n do x plus (next; "0123456789" ! "0"); l);
           (?"d" | for i to n do x plus (next; "0123456789" ! "0"); l));
          proc string mould = void: while? "a" do to n do x plus
                (supp | "." | next);
     tis: (frame[1] = 6 | string mould | : intreal mould; frame[1] > 3 |
            "⊥" ! "⊥"; intreal mould);
          t := x
    end;
```

10.5.3.2. continued 2

```
g) proc % trans indit choice =
     (file f, ref int c, string format, ref int p) :
     (req insertion(f, format, p) ; p plus 2 ; c := 1 ;
      while ¬ ask lit(f, format, p) do
      (c plus 1 ; format[p] = "," | p plus 1 | undefined) ;
      while format[p] ≠ ")" do(p plus 1 ; skip lit(format, p)) ;
    p plus 1 ; req insertion(f, format, p)) ;
h) proc % trans indit bool =
     (file f, ref bool b, string format, ref int p) :
     (req insertion(f, format, p) ; (format[p + 1] = "(" |
       p plus 2 ; (b := ask lit(f, format, p) |
         p plus 1 ; skip lit(format, p) |:
         p plus 1; ¬ ask lit(f, format, p) | undefined) |
       char k ; get string(f, k) ; b := (k = "1" | true |:
         k = "0" | false)) ;
    p plus 1 ; req insertion(f, format, p)) ;
i) proc % req insertion = (file f, [1 : int l false] char s, ref int p) :
    while(p > l | false |: do align(f, s, p) | true |
      req lit(f, s, p)) do skip ;
j) proc % req lit = (file f, string s, ref int p) bool :
    (int q = p ; int n ; replicator(s, p, n) ;
         (s[p] = """" | int r = p ; to n do(p := r ;
          while(s[p plus 1] = """" | s[p plus 1] = """" | true) do
          (char k ; get string(f, k) ; k ≠ s[p] | undefined)) ; true |
                             p := q ; false)) ;

k) proc % ask lit = (file f, string s, ref int p) bool :
    (int c = char of f ; int n ; replicator(s, p, n) ;
     (s[p] = """" | int r = p ; to n do(p := r ;
     while(s[p plus 1] = """" | s[p plus 1] = """" | true) do
     (char k ; get string(f, k) ; k ≠ s[p] | l)) ; true.
    l: while(s[p plus 1] = """" | s[p plus 1] = """" | true) do skip ;
      char of f := c ; false)) ;
```

10.5.4. Binary transput

```
a) proc % to bin = (file f, simplout x) [] int :
     c a value of mode 'row of integral' whose lower bound is one,
       and whose upper bound depends on the value of 'f' and on the
       mode of the value of 'x'; furthermore,
       x = from bin(f, to bin(f, x)) c ;
b) proc % from bin = (file f, [] int y) simplout :
     c a value, if one exists, of a mode from which that specified by
       simplout is united, such that y = to bin(f, from bin(f, y)) c ;
```

{On some channels a more straightforward way of transput is available. Some properties of this binary transput depend on the particular implementation}

10.5.4.1. Binary output

```
a) proc write bin = ([] outtype x) : put bin(stand back, x) ;
b) proc put bin = (file file, [1 : int n] outtype x) :
     if bin possible[chan of file] ∧ opened of file
     then if ¬ set possible[chan of file] thef state def of file
          then(state get of file ∨ ¬ state bin of file | undefined)
          else state def of file := state bin of file := true ;
               state get of file := false
          fi ;
          for k to n do
          ([1 : int l] simplout y = straightout x[k] ;
          for j to l do
          ([1 : int m] int bin = to bin(file, y[j]) ;
          for i to m do(next plc(file) ;
          book of bfile of file[page of file, line of file,
                char of file] := bin[i])))
     else undefined
     fi ;
```

10.5.4.2. Binary input

```
a) proc read bin = ([] intype x) : get bin(stand back, x) ;
b) proc get bin = (file file, [1 : int n] intype x) :
     if bin possible[chan of file] ∧ opened of file
     then if ¬ set possible [chan of file] thef state def of file
          then(¬state get of file ∨ ¬ state bin of file | undefined)
          else state def of file := state bin of file :=
                 state get of file := true
          fi ;
          for k to n do
          ([1 : int l] ref simplout y = straightin x[k] ;
          for j to l do
          ([1 : int m] int bin := to bin(file, y[j]); simplout r ;
          for i to m do(next plc(file) ;
          bin[i] := book of bfile of file[page of file, line of file,
            char of file]) ;
          r := from bin(file, bin) ;
          (¢ (ref L int ii ; (ii ::= y[j] |:
             val ii ::= r | l | undefined)) ¢) ;
          (¢ (ref L real xx ; (xx ::= y[j] |:
             val xx ::= r | l | undefined)) ¢) ;
          (¢ (ref L compl zz ; (zz ::= y[j] |:
             val zz ::= r | l | undefined)) ¢) ;
          (ref string ss ; (ss ::= y[j] |:
             val ss ::= r | l | undefined)) ;
          (ref char cc ; (cc ::= y[j] |: val cc ::= r | l | undefined)) ;
          (ref bool bb ; (bb ::= y[j] |: val bb ::= r | l | undefined)) ;
          l : skip))
     else undefined
     fi ;
```

{But Eeyore wasn't listening. He was
taking the balloon out, and putting it
back again, as happy as could be. ...
Winnie-the-Pooh, A.A. Milne.}

11. Examples

11.1. Complex square root

A declaration in which *compsqrt* is a procedure-with-[complex]-parameter-[complex]-identifier (Here [complex] stands for structured-with- real-named-letter-r-letter-e-and-real-named-letter-i-letter-m.) :

a) **proc** *compsqrt* = (**compl** *z*) **compl** : **c** *the square root whose real part is nonnegative of the complex number z* **c**

b) **begin** **real** *x* = **re** *z*, *y* = **im** *z* ;

c) **real** *rp* = *sqrt*((**abs** *x* + *sqrt*(*x* ↑ 2 + *y* ↑ 2))/2) ;

d) **real** *ip* = (*rp* = 0 | 0 | *y*/(2 × *rp*)) ;

e) (*x* ≥ 0 | (*rp* ⊥ *ip*) | (**abs** *ip* ⊥ (*y* ≥ 0 | *rp* | -*rp*)))

f) **end** *compsqrt*

[complex]-clause-calls {8.6.3} using *compsqrt*:

g) *compsqrt*(*w*)

h) *compsqrt*(-3.14)

i) *compsqrt*(-1)

11.2. Innerproduct1

A declaration in which *innerproduct1* is a procedure-with-integral-parameter-and-procedure-with-integral-parameter-real-parameter-and-procedure-with-integral-parameter-real-parameter-real-identifier:

```
a) proc innerproduct1 = (int n, proc(int) real x, y) real :
   comment the innerproduct of two vectors, each with n components,
   x(i), y(i), i = 1, ..., n, where x and y are arbitrary mappings
   from integer to real number comment
b) begin long real s := long 0 ;
c) for i to n do s plus leng x(i) × leng y(i) ;
d) short s
e) end innerproduct1
```

Real-clause-calls {8.6.3} using *innerproduct1*:

```
f) innerproduct1(m, (int j) real : x1[j], (int j) real : y1[j])
g) innerproduct1(n, nsin, ncos)
```

11.3. Innerproduct2

A declaration in which *innerproduct2* is a procedure-with-reference-to-row-of-real-parameter-and-reference-to-row-of-real-parameter-real-identifier:

```
a) proc innerproduct2 = (ref[1 : int n] real a, b) real :
    c the innerproduct of two vectors a and b with n elements c
b) begin long real s := long 0 ;
c) for i to n do s plus leng a[i] × leng b[i] ;
d) short s
e) end innerproduct2
```

Real-clause-calls using *innerproduct2*:

```
f) innerproduct2(x1, y1)
g) innerproduct2(y2[2], y2[, 3])
```

11.4. Innerproduct3

A declaration in which *innerproduct3* is a procedure-with-reference-to-integral-parameter-and-integral-parameter-and-procedure-real-parameter-and-procedure-real-parameter-real-identifier:

```
a) proc innerproduct3 = (ref int i, int n, proc real xi, yi) real :
      comment the innerproduct of two vectors whose n elements are the
      values of the expressions xi and yi and which depend, in general,
      on the value of i comment
b) begin long real s := long 0 ;
c) for k to n do(i := k ; s plus leng xi × leng yi) ;
d) short s
e) end innarproduct3
```

A real-clause-call using *innerproduct3*:

```
f) innerproduct3(j, 8, x1[j], y1[j + 1])
```

11.5. Largest element

A declaration in which *absmax* is a procedure-with-reference-to-row-of-row-of-real-parameter-and-reference-to-real-parameter-and-reference-to-integral-parameter-and-reference-to-integral-parameter-identifier:

```
a) proc absmax = (ref[1 : int m, 1 : int n] real a,
b) c result c ref real y, c subscripts c ref int i, k) :
      comment the absolute value of the element of greatest absolute value
      of the m by n matrix a is assigned to y, and the subscripts of this
      element to i and k comment
c) begin y := -1 ;
d) for p to m do for q to n do
e) if abs a[p, q] > y then y := abs a[i := p, k := q] fi
f) end absmax
```

Void-clause-calls {8.6.3} using *absmax*:

```
g) absmax(x2, x, i, j)
h) absmax(x2, x, loc int, loc int)
```

11.6. Euler summation

```
a) proc euler = (proc(int) real f, real eps, int tim) real :
     comment the sum for i from 1 to infinity of f(i), computed by means
     of a suitably refined euler transformation. The summation is
     terminated when the absolute values of the terms of the transformed
     series are found to be less than eps tim times in succession. This
     transformation is particularly efficient in the case of a slowly
     convergent or divergent alternating series comment
b) begin int n := 1, t; real mn, ds := eps;  [1 : 16] real m ;
c)       real sum := (m[1] := f(1))/2 ;
d)       for i from 2 while(t := (abs ds < eps | t + 1 | 1)) ≤ tim do
e)         begin mn := f(i) ;
f)           for k to n do begin mn := ((ds := mn) + m[k])/2 ;
g)                               m[k] := ds end ;
h)           sum plus(ds := (abs mn < abs m[n] ∧ n < 16 |
i)                         n plus 1 ; m[n] := mn ; mn/2 | mn))
j)         end ;
k)       sum
l) end euler
```

A clause-call using *euler*:

```
m) euler((int i) real : (odd i | -1/i | 1/i), 1₁₀-5, 2)
```

11.7. The norm of a vector

```
a) proc norm = (ref[1 : int n] real a) real :
     c the euclidean norm of the vector a with n elements c
b)     (long real s := long 0 ;
c)      for k to n do s plus leng a[k] ↑ 2 ;
d)      short long sqrt(s))
```

For a use of *norm* as a clause-call, see 11.8.d.

11.8. Determinant of a matrix

```
a)  proc det = (ref[1 : int n, 1 : int n] real a,
b)                        ref[1 : int n] int p) real :
      comment the determinant of the square matrix a of order n by the
      method of Crout with row interchanges: a is replaced by its triangular
      decomposition l × u with all u[k, k] = 1. The vector p gives as
      output the pivotal row indices; the k-th pivot is chosen in the k-th
      column of l such that abs l[i, k]/row norm is maximal comment
c)      begin[1 : n] real v; real d := 1, r := -1, s, pivot ;
d)      for i to n do v[i] := norm(a[i]) ;
e)      for k to n do
f)        begin int k1 = k - 1 ; ref int pk = p[k] ;
g)        ref[,] real al = a[, 1 : k1], au = a[1 : k1] ;
h)        ref[] real ak = a[k], ka = a[, k], apk = a[pk],
i)          alk = al[k], kau = au[, k] ;
j)        for i from k to n do
k)          begin ref real aik = ka[i] ;
l)          if(s := abs(aik minus innerproduct 2(al[i], kau))/v[i]) > r
m)            then r := s ; pk := i fi
n)          end for i ;
o)        v[pk] := v[k] ; pivot := ka[pk] ;
p)        for j to n do
q)            begin ref real akj = ak[j], apkj = apk[j] ;
r)            r := akj ; akj := if j ≤ k then apkj
s)            else(apkj - innerproduct2(alk, au[, j]))/pivot fi ;
t)            if pk ≠ k then apkj := -r fi
u)            end for j ;
v)        d times pivot
w)        end for k ;
x)      d
y)      end det
```

A clause-call using *det*:

```
z)      det(y2, i1)
```

11.9. Greatest common divisor

An example of a recursive procedure:

```
a) proc gcd = (int a, b) int :
      c the greatest common divisor of two integers c
b)      (b = 0 | abs a | gcd(b, a ÷: b))
```

An expression-call using *gcd*:

```
c)      gcd(n, 124)
```

11.10. Continued fraction

An example of a recursive operation:

```
a) op / = ([1 : int n] real a, b) real :
      comment the value of a/b is that of the continued fraction
      a1/(b1 + a2/(b2 + ... an/bn)...) comment
b)      (n = 1 | a[1]/b[1] | a[1]/(b[1] + a[2 : ]/b[2 : ]))
```

A formula using /:

c) *x1/y1*

{The use of recursion may often be elegant rather than efficient as in 11.9 and 11.10. See, however, 11.11 for an example in which recursion is of the essence.}

11.11. Formula manipulation

```
a) begin union form = (ref const, ref var, ref triple, ref call);
b) struct const = (real value);
c) struct var = (string name, real value);
d) struct triple = (form left operand, int operator, form right operand);
e) struct function = (ref var bound var, form body);
f) struct call = (ref function function name, form parameter);
g) int plus = 1, minus = 2, times = 3, by = 4, to = 5;
h) const zero, one; value of zero := 0; value of one := 1;
i) op = = (form a, ref const b) bool :
      (ref const ec ; (ec ::= a | val ec :=: b | false)) ;
j) op + = (form a, b) form :
      (a = zero | b |: b = zero | a | triple := (a, plus, b));
k) op - = (form a, b) form : (b = zero | a | triple := (a, minus, b));
l) op × = (form a, b) form :
      (a = zero ∨ b = zero | zero |: a = one | b |: b = one | a |
                                        triple := (a, times, b));
m) op / = (form a, b) form :
      (a = zero ∧ ¬ b = zero | zero |: b = one | a | triple := (a, by, b));
n) op ↑ = (form a, ref const b) form :
      (a = one ∨ b :=: zero | one |: b :=: one | a | triple := (a, to, b));
o) proc derivative of = (form e, c with respect to c ref var x) form :
p) begin ref const ec ; ref var ev ; ref triple et ; ref call ef ;
q) if   ec :: e then zero
r) elsf ev ::= e then ev :=: x | one | zero)
s) elsf et ::= e then
t)          form u = left operand of et, v = right operand of et,
u)          udash = derivative of (u, c with respect to c x),
v)          vdash = derivative of (v, c with respect to c x) ;
w)          case operator of et in
x)            udash + vdash, udash - vdash,
y)            u × vdash + udash × v, (udash - et × vdash)/v,
z)          (ec ::=v| v × u ↑ (const c; value of c := value of ec - 1; c) × udash
            esac
```

11.11. continued

```
aa) elsf ef ::= e then
ab)         ref function f = function name of ef;
ac)         form g = parameter of ef;
ad)         ref var y = bound var of f;
ae)         function fdash := (y, derivative of(body of f, y));
af)         (call := (fdash, g)) × derivative of(g, x)
ag) fi
ah) end derivative;
ai) proc value of = (form e) real :
aj)   begin ref const ec ; ref var ev ; ref triple et ; ref call ef;
ak)   if   ec ::= e then value of ec
al)   elsf ev ::= e then value of ev
am)   elsf et ::= e then
an)           real u = value of(left operand of et),
ao)                v = value of(right operand of et);
ap)           case operator of et in
aq)             u + v, u - v, u × v, u / v, exp(v × ln(u)) esac
ar)   elsf ef ::= e then
as)           ref function f = function name of ef;
at)           value of bound var of f := value of(parameter of ef);
au)           value of(body of f)
av)   fi
aw)   end value of;
ax)   form f, g ; var a := ("a", skip), b := ("b", skip), x := ("x", skip);
ay)   start here:
az)   read((value of a, value of b, value of x));
ba)   f := a + x / (b + x) ; g := (f + one) / (f - one);
bb)   print((value of a, value of b, value of x,
              value of(derivative of(g, c with respect to c x)))
bc) end example
```