MR 92

A Draft Proposal for the Algorithmic Language ALGOL 68

A. van Wijngaarden
with the assistance of
B.J. Mailloux
and
J.E.L. Peck

January 1968

0. Introduction

0.1. Aims and principles of design

a) In defining the Algorithmic Language ALGOL 68, the members of Working Group 2.1 of the International Federation for Information Processing express their belief in the value of a common programming language serving many people in many countries.

b) The language is designed to communicate algorithms, to execute them efficiently on a variety of different computers, and to aid in teaching them to students.

c) The members of the Group, influenced by several years of experience with ALGOL 60 and other programming languages, hope that the following has been achieved:

0.1.1. Completeness and clarity of description

The Group wishes to contribute to the solution of the problems of describing a language clearly and completely. It is recognized, however, that the method adopted in this Report may be difficult for the uninitiated reader.

0.1.2. Orthogonal design

The number of independent primitive concepts was minimized in order that the language be easy to describe, to learn, and to implement. On the other hand, these concepts have been applied "orthogonally" in order to maximize the expressive power of the language, and yet without introducing deleterious superfluities.

0.1.3. Security

ALGOL 68 has been designed in such a way that nearly all syntactical and many other errors can be detected easily before they lead to calamitous results. Furthermore, the opportunities for making such errors are greatly restricted.

0.1.4. Efficiency

ALGOL 68 allows the programmer to specify programs which can be run efficiently on present-day computers and yet do not require sophisticated and time-consuming optimization features of a compiler; see e.g. 11.8.

0.1.4.1. Static mode checking

The syntax of ALGOL 68 is such that no mode checking during run time is necessary except during the elaboration of conformity-relations {8.1.2} the use of which is required only in those cases in which the programmer explicitly makes use of the flexibility offered by the united mode feature.

0.1.4.2. Independent compilation

ALGOL 68 has been designed such that the main line programs and procedures can be compiled independently of one another without loss of object program efficiency, provided that during each such independent compilation, specification - by means not given in this Report - of the mode of all nonlocal quantities is provided.

0.1.4.3. Loop optimization

Iterative processes are formulated in ALGOL 68 in such a way that straightforward application of well-known optimization techniques yields large gains during run time without excessive increase of compilation time.

0.2. Comparison with ALGOL 60

a) ALGOL 68 is a language of wider applicability and power than ALGOL 60. Although influenced by the lessons learned from ALGOL 60, ALGOL 68 has not been designed as an expansion of ALGOL 60 but rather as a completely new language based on new insights into the essential, fundamental concepts of computing and a new description technique.

0.2. continued

b) The result is that the successful features of ALGOL 60 reappear in ALGOL 68 but as special cases of more general constructions, along with completely new features. It is, therefore, difficult to isolate differences between the two languages; however, the following sections are intended to give insight into some of the more striking differences.

0.2.1. Values in ALGOL 68

a) Whereas ALGOL 60 has values of the types integer, real, boolean and string, ALGOL 68 features an infinity of "modes", i.e. generalizations of the concept type.

b) Each plain value is either arithmetic, i.e. of integral or real mode and then it is of one of several lengths, or it is of boolean, character or pattern mode.

c) In ALGOL 60 composition of values is possible into arrays, whereas in ALGOL 68, in addition to such "multiple" values, also "structured" values, composed of values of possibly different modes, are defined and manipulated. An example of a multiple value is a character array, which corresponds approximately to the ALGOL 60 string; examples of structured values are complex numbers and symbolic formulae.

d) In ALGOL 68 the concept of a "name" is introduced, i.e. a value which is said to "refer to" another value; such a name-value pair corresponds to the ALGOL 60 variable. However, any name may take the value position in a name-value pair and thus chains of indirect addresses can be built up.

e) The ALGOL 60 concept of a procedure body is generalized in ALGOL 68 to the concept "routine", which also includes the formal parameters, and which is itself a value and therefore can be manipulated like any other value.

0.2.1. continued

f) In contrast with plain values and multiple and structured values composed of plain values only, the significance of a name or routine or a multiple or structured value composed of names or routines, possibly amongst other values, is, in general, dependent on the context in which it appears. Therefore, the use of names and routines as values is subject to some natural restrictions related to their "scope".

0.2.2. Declarations in ALGOL 68

a) Whereas ALGOL 60 has type declarations, array declarations, switch declarations and procedure declarations, ALGOL 68 features the "identity-declaration" whose expressive power includes all of these, and more. In fact, the identity-declaration declares not only variables, but also constants, of any mode and, moreover, forms the basis of a highly efficient and powerful parameter mechanism.

b) Moreover, in ALGOL 68, a "mode-declaration" permits the construction of new modes from already existing ones. In particular, the modes of multiple values and structured values may be defined this way; in addition a union of modes may be defined for use in an identity-declaration allowing each value referred to by a given name to be of one of the constituent modes.

c) Finally, in ALGOL 68, a "priority-declaration" and an "operation-declaration" permit the introduction of new operators, the definition of their operation and the extension or revision of the class of operands applicable to already established operators.

0.2.3. Dynamic storage allocation in ALGOL 68

Whereas ALGOL 60 (apart from the so-called "own dynamic arrays") implies a "stack"-oriented storage-allocation regime, sufficient to cope with a statically (i.e. at compile time) determined number of values, ALGOL 68 provides, in addition, the ability to generate a dynamically (i.e. at run time) determined number of values, which ability implies the use of additional, well established, storage-allocation techniques.

0.2.4. Collateral elaboration in ALGOL 68

Whereas, in ALGOL 60, statements are "executed consecutively", in ALGOL 68 "phrases" are "elaborated serially" or "collaterally". This last facility is conducive to more efficient object programs under many circumstances, and increases the expressive power of the language. Facilities for parallel programming, though restricted to the essentials in view of the none-too-advanced state of the art, have been introduced.

0.2.5. Standard declarations in ALGOL 68

The ALGOL 60 standard functions are all included in ALGOL 68 along with many other standard declarations. Amongst these are "environment enquiries", which make it possible to determine certain properties of an implementation, and "input-output" declarations, which make it possible, at run time, to obtain data from and to deliver results to external media.

0.2.6. Some particular constructions in ALGOL 68

a) The ALGOL 60 concepts of block, compound statement and parenthesized expression are unified in ALGOL 68 into "closed-clause". A closed-clause may be an expression and possess a value. Similarly, the ALGOL 68 "assignation", which is a generalization of the ALGOL 60 assignment statement, may be an expression and, as such, also possesses a value.

0.2.6. continued

b) The ALGOL 60 concept of subscription is generalized to the ALGOL 68 concept of "indexing", which allows the selection not only of a single element of an array but also of subarrays with the same or any smaller dimensionality and with possibly altered bounds.

c) ALGOL 68 provides not only the multiple values mentioned in 0.2.1.c, but also "collateral-expressions" which serve to compose these values from other, simpler values.

d) The ALGOL 60 for statement is modified into a more concise and efficient "repetitive statement".

e) The ALGOL 60 conditional expression and conditional statement, unified into a "conditional-clause", are improved by requiring them to end with a closing symbol whereby the two alternative clauses admit the same syntactic possibilities. Moreover, the conditional-clause is generalized into a "case clause" which allows the efficient selection from an arbitrary number of clauses depending on the value of an integral expression.

f) Some less successful ALGOL 60 concepts, such as own quantities and integer labels have not been included in ALGOL 68, and some concepts like designational expressions and switches do not appear as such in ALGOL 68, but their expressive power is included in other, more general, constructions.

1. Language and metalanguage

1.1. The method of description

1.1.1. The strict, extended and representation languages

a) The algorithmic language ALGOL 68 is a language in which algorithms can be described for execution by means of a computer, i.e. either an automaton or a human being. It is defined in three stages called 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, the typographical marks of this report are used. These bear no relation to the typographical marks 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", i.e. nonempty sequences of small letters, possibly interspersed with nonsignificant blanks and/or hyphens.

b) A "list of notions" either is empty, or is a notion, or consists of a list of notions followed by a comma followed by a notion.

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

d) A "symbol" is a notion ending with 'symbol'.

1.1.2. continued

e) A "production" of a notion is either a direct production of it or a list of notions obtained by replacing in a production of that notion a notion by a direct production of that notion.

f) A "terminal production" of a notion is a production of that notion which consists of symbols and commas only.

{The production rule of the strict language
variable-point numeral : integral part option, fractional part.
(5.1.2.1.b) contains a direct production
integral part option, fractional part
of the notion 'variable-point numeral'. A terminal production of this same notion is
zero symbol, point symbol, one symbol.
The notion 'zero symbol' is an example of a symbol. The line "twas brillig and the slithy toves" is not a relevant notion of the strict language, in that it does not end with 'symbol' and no production rule for it is given (1.1.5 Step 3).}

1.1.3. The syntax of the metalanguage

a) The production rules of the strict language are partly defined by enumeration and are partly generated with the aid of a metalanguage whose syntax consists of a set of production rules for "metanotions", i.e. nonempty sequences of capital letters.

b) A "list of metanotions" either is empty or is a notion, or consists of one or more metanotions separated, and possibly preceded and/or followed, by notions and/or blanks.

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

1.1.3. continued

d) A production of a metanotion is either a direct production of that metanotion or a list of metanotions obtained by replacing in a production of that metanotion a metanotion by a direct production of that metanotion.

e) A terminal production of a metanotion is a production of that metanotion which is a notion.

{The production rule
TAG : LETTER.
derived from 1.2.1.1 contains a direct production 'LETTER' of the metanotion 'TAG'. A particular terminal production of the metanotion 'TAG' is the notion 'letter x symbol' (see 1.2.1.m, n). The production rule
EMPTY: . (1.2.1.i)
has an empty direct production. }

1.1.4. The production rules of the metalanguage

a) In the rule beginning with 'ALPHA' {1.2.1.n}, the point may be replaced arbitrarily often by a semicolon followed by an other letter followed by a point.

b) The production rules of the metalanguage are the rules obtained from the rules in Section 1.2 as follows:
Step: If some rule contains one or more semicolons, then it is replaced by two new rules, the first 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, the Step is taken again.

{For instance, the rule

1.1.4. continued

TAG : LETTER ; TAG LETTER ; TAG DIGIT.
from 1.2.1.1 is replaced by the rules
TAG : LETTER.
TAG : TAG LETTER ; TAG DIGIT.
and the second of these is replaced by
TAG : TAG LETTER.
TAG : TAG DIGIT.
thus resulting in three rules from the original one.

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

1.1.5. The production rules of the strict language

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 the Step of 1.1.4.b ;
Step 2: If a given rule now contains one or more sequences of capital letters, then this (these) sequence(s) is (are) interpreted as (a) sequence(s) of metanotions occurring in Section 1.2 {The metanotions of 1.2 have been chosen such that this interpretation is unique.}, and then for each terminal production of such a metanotion, a new rule is obtained by replacing, in a copy of the given rule, all occurrences of that metanotion by that terminal production, whereupon the given rule is discarded and Step 2 is taken; otherwise, the given rule is a production rule of the strict language.
Step 3: A number of production rules for the notions 'other character token' and 'other multicharacter token' each of whose direct productions is a symbol different from any other symbol may be added.
{The rule
actual LOWPER bound : strict LOWPER bound.
derived from 7.1.1.r by Step 1 is used in Step 2 to provide two production rules of the strict language, viz.
actual lower bound : strict lower bound.
actual upper bound : strict upper bound. .

1.1.5. continued

Note that
actual lower bound : strict upper bound.
is not a production rule of the strict language, since the replacement of the metanotion 'LOWPER' by one of its productions must be consistent at each occurrence. 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.
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 notion to the right of the colon even though it is not a symbol. }

1.1.6. The semantics of the strict language

a) A terminal production of a notion is considered as a linearly ordered sequence of symbols. This order, which goes from "left" to "right", 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) sequence(s) of symbols "contains" that second sequence of symbols.

c) Unless otherwise specified {d}, a "paranotion" at an occurrence not under "Syntax", not between apostrophes and not within another paranotion stands for any terminal production of some notion; a paranotion being either
i) a production of 'token' {3.0.1.a} ending with 'symbol', in which

case it then stands for itself {e.g. "begin symbol"}, or

ii) a notion whose production rule(s) do(es) not begin with an asterisk, in which case it then stands for any terminal production of itself {e.g. "basic-token" (3.0.1.b) stands for any terminal production of 'letter token', 'denotation token', etc.}, or

iii) a notion whose production rule(s) do(es) begin with an asterisk, in which case it then stands for any terminal production of any of its direct productions {e.g. "token" (3.0.1.a) stands for any terminal production of 'basic token', 'special token' or 'other token'.}, or

iv) a paranotion followed by the letter "s", or a paranotion ending with the letter "y" in which that last letter has been replaced by the letters "ies", in which case it then stands for some number of the terminal productions stood for by that paranotion {e.g. "basic-tokens" stands for some number of terminal productions of 'letter token' and/or 'denotation token', etc., and "primaries" stands for some number of terminal productions stood for by 'primary'.}, or

v) a paranotion whose first letter has been replaced by the corresponding capital letter, in which case it then stands for the terminal productions stood for by that paranotion before the replacement {e.g. "Basic-tokens" stands for what "basic-tokens" stands for}, or

vi) a paranotion in which a terminal production of 'MODE' has been omitted, in which case it then stands for any terminal production stood for by any paranotion from which the given paranotion could be obtained by omitting a terminal production of 'MODE' {e.g. "slice" stands for any terminal production of "MODE slice" (8.4.1.a), where "MODE" stands for any terminal production of the metanotion 'MODE'.}.

{As an aid to the reader, paranotions are printed in a different font, and, when not under Syntax or between apostrophes, with hyphens instead of spaces. Moreover, as an additional aid, a number of superfluous rules beginning with an asterisk have been included. }

d) When a paranotion is said to be a "constituent" of a second paranotion, then the first paranotion stands for any terminal production stood for

by it according to 1.2.6.c which is contained in a terminal production stood for by the second paranotion but not contained in a terminal production stood for by either of these paranotions contained in that second terminal production.

{e.g. *j* := *1* is a constituent assignation (8.8.1) of the assignation *i* := *j* := *1*, but not of the serial-statement (6.1.1.b) *i* := *j* := *1* ; *k* := *2*.}

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

f) If a given sequence of symbols is a production of a given notion which is itself a production of another notion, then, except as otherwise specified, its elaboration as terminal production of that other notion consists of its elaboration as terminal production of the given notion.

{e.g. the elaboration of *random* as a fitted-real-cohesion is its elaboration as a called-real-cohesion (8.2.0.1.e).}

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

h) This Report (possibly along with such information from outside it) associates a meaning specifically with programs {2.1.a} satisfying the "context conditions" {4.4}.

1.1.7. The extended language

The extended language encompasses the strict language; i.e. a program in the strict language, possibly subjected to a number of notational changes by virtue of "extensions" given in Chapter 9 is a program in the extended language and has the same meaning.

{e.g. *real x, y, z* means the same as *(real x, real y, real z)*; see 9.2.}

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 {commas, not comma-symbols} are deleted, is a program in the representation language and has the same meaning.

b) Each version of the language in which representations are used which are sufficiently close to the given representations to be recognized 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 but bear a one-to-one relationship with them, 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}.

1.2. The metaproduction rules

1.2.1. Metaproduction rules of modes

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

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

1.2.2. Metaproduction rules associated with modes

a) PRIMITIVE : integral ; real ; boolean ; character ; pattern.
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 a FIELDS ; row of MODE ;
reference to NONPROC ; UNITED.
i) LMODE : MODE.
j) RMODE : MODE.
k) LMODES : MODES.
l) RMODES : MODES.
m) LFIELDS : FIELDS.
n) RFIELDS : FIELDS.
o) COMPLEX : structured with a real named letter r symbol
letter e symbol and a real named letter i symbol
letter m symbol.
p) STRING : row of character.
q) BITS : row of boolean.
r) OTHER : boolean ; COMPLEX ; STRING.
s) MABEL : MODE ; label.

1.2.3. Metaproduction rules associated with phrases

a) PHRASE : declaration ; CLAUSE.
b) CLAUSE : statement ; MODE expression.
c) SOME : serial ; unitary ; CLOSED ; choice ; THELSE.
d) THELSE : then ; else.
e) CLOSED : closed ; collateral ; conditional.
f) COERCETY : COERCED ; EMPTY.
g) COERCED : adapted ; adjusted ; fitted ; called ; peeled.

1.2.4. Metaproduction rules associated with formulas

a) OPERAND : MODE FORM.
b) FORM : PRIORITY ADIC formula ; cohesion ; assignation.
c) PRIORITY : priority NUMBER.
d) NUMBER : zero ; ONE ; TWO ; THREE ; FOUR ; FIVE ; SIX ;
SEVEN ; EIGHT ; NINE.
e) ONE : zero plus one.
f) TWO : ONE plus one.
g) THREE : TWO plus one.
h) FOUR : THREE plus one.
i) FIVE : FOUR plus one.
j) SIX : FIVE plus one.
k) SEVEN : SIX plus one.
l) EIGHT : SEVEN plus one.
m) NINE : EIGHT plus one.
n) ADIC : monadic ; dyadic.
o) OPERATOR : RMODE MODE PRIORITY monadic operator ;
LMODE RMODE MODE PRIORITY dyadic operator.

1.2.5. Other metaproduction rules

a) VICTAL : virtual ; actual ; formal.
b) LOWPER : lower ; upper.
c) LIGHT : loose ; tight.
d) ANY : sign ; zero ; digit ; point ; exponent ; complex ; string.
e) NOTION : ALPHA ; NOTION ALPHA.
f) SEPARATOR : comma symbol ; EMPTY ; go on symbol ; completer ; sequencer.

{Rule e implies that all notions (1.1.2.a) are productions (1.1.3.d) of the metanotion (1.1.3.a) 'NOTION'. For the use of this metanotion, see 3.0.1.e, f, g, h, i. }

{"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.

{Some of these pragmatic remarks are examples written in the reference language. In these examples, identifiers are used out of context from their defining occurrences. Unless otherwise specified, these identifiers identify those in the identity-declarations of the standard-declarations in Chapter 10 (e.g. *random* from 10.3.k or *pi* from 10.3.a) 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 ref real x or y = expr(random < .5 | x | y) ;
proc real ncos = (int i) : cos(2 × pi × i/n) ;
proc real nsin = (int i) : sin(2 × pi × i/n) ;
proc real g = (real u) : (arctan(u) - a + u - 1) ;
exit: princeton: grenoble: kootwijk: warsaw: zandvoort: amsterdam: x := 1.}
```

2. The computer and the program

2.1. Syntax

a) program : open, standard declarations, go on symbol,
library declarations option, particular program, close.
b) standard declarations : serial declaration.
c) library declarations : serial declaration, go on symbol.
d) particular program : label sequence option, CLOSED statement.

{For standard-declarations see Chapter 10 and for closed-statements see 6.4. The specification of library-declarations is undefined. }

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 defined in terms of a hypothetical "computer" which performs a set of "actions" {2.2.7}, 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 terminal productions {1.1.2.f} of notions, at different "occurrences". Internal objects are "values" {2.2.3}, at different "instances".

2.2.2. Relationships

a) Relationships are either "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.

2.2.2. continued 1

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

c) An "identifier" {4.1}, "indication" {4.2} or "operator" {4.3} at a given occurrence may identify the same identifier, indication or operator at a "defining" occurrence {4.1.2.a, 4.2.2.a, 4.3.2.a}.

d) The relationships between an external object and an internal object are: "to denote" and "to possess".

e) A "denotation" {5} denotes a value; this relationship is permanent.

f) An identifier may denote a value and an operator may denote {more specifically} an "operation" {2.2.5}. This relationship is caused to hold by the elaboration of an "identity-declaration" {7.4} or operation-declaration" {7.5}, respectively, and ceases to hold upon the end of the elaboration {6.4.2} of the smallest range {4.1.1.e} containing that declaration.

g) 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}. This relationship is caused to hold by the elaboration of the external object as terminal production of the given notion, and continues to hold until the next elaboration, if any, of the same occurrence of that external object is initiated, whereupon it ceases to hold.

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.

2.2.2. continued 2

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" {8.8.2.a} of that value to that name and continues to hold until an instance of another 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 either
  "plain" values {2.2.3.1}, which are independent of the program and its elaboration,
or "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,
or "routines" {2.2.3.4}, which are certain sequences of symbols defined by the program,
or names {2.2.3.5}, which are created by the elaboration of the program.

2.2.3.1. Plain values

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

2.2.3.1. continued

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, called 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}, and, moreover, a third integer (real number) of that length number may exist, called the first integer (real number) minus the other one {10.2.2.g, 10.2.3.g}. Finally, for each pair of real numbers of the same length number, two real numbers of that length number may exist, called the first real number times (divided by) the other one {10.2.3.1, m}.

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" 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 an integer of the same length number or on those values 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; this relationship is defined only in so far that different characters have different integral equivalents.

g) A pattern is equivalent to a row of characters.

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 {8.6.2. Step 2} by a specific field-selector {7.1.1.e}.

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 {8.4.2. Step 7} by a unique integer, its "index".

b) The descriptor, which "describes" the elements, 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 called 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 any $l_i > u_i$, then the number of elements in the multiple value is zero; otherwise, it is

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

{These data suffice, given an n-tuple $(r_1, \ldots, r_n)$ of integers satisfying $l_i \leq r_i \leq u_i$, to compute a unique index from

$$c + (r_1 - l_1) \times d_1 + \ldots + (r_n - l_n) \times d_n.$$

In a given instance of a multiple value, a state which is 0(1) indicates that the given value can (cannot) be "superseded" (2.2.3.5.d) by an instance of a multiple value in which the bound corresponding to the state differs from that in the given value. }

c) A subvalue of a given multiple value is a multiple value composed of a descriptor which describes a subset of the elements of the given multiple value, and that subset.

2.2.3.4. Routines

A routine is a token-sequence which is the same as some closed-clause.

2.2.3.5. Names

a) There is one name, called nil, whose scope {2.2.4.2} is the program and which does not refer to any value. Any other name is created by the elaboration of an actual-declarer {7.1.2.c. Step 7, and refers to precisely one (instance of a) value}

b) If a given name refers to a structured value, then to each of its fields there refers a specific name which is uniquely determined by the given name and the field-selector selecting that field, and whose scope {2.2.4.2} is that of the given name.

c) If a given name refers to a given multiple value, then to each of its subvalues and elements there refers a specific name which is uniquely determined by the given name and the descriptor of that subvalue or the index of that element, and whose scope {2.2.4.2} is that of the given name.

d) 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.

e) A value ceases to be referred to by a given name when another value is caused to be referred to by that name.

2.2.4. Modes and scopes

2.2.4.1. Modes

a) Each value is of one specific mode, i.e. a terminal production of 'MODE' {1.2.1.a}.

b) The mode of an integer of length number one is 'integral', that of a real number of length number one is 'real', that of a truth value is 'boolean', that of a character is 'character', and that of a pattern is 'pattern'.

c) The mode of an integer or real number of length number n > 1 is 'integral' or 'real' respectively, preceded by (n - 1) times 'long'.

d) The mode of a structured value is 'structured with' followed by one or more "field modes" separated by 'and', one corresponding to each field taken in the same order, each field mode being 'a' followed by the mode of the field followed by 'named' followed by the field-selector which selects that field. If two of these field-selectors are the same, then the mode is "enigmatic" {see 4.4.2}.

{e.g. The mode specified by *struct(int i, real i)* is enigmatic, but that specified by *struct(int i, struct(int i, j) j)* is not. }

e) The mode of a multiple value is the mode of its elements preceded by as many times 'row of' as there are quintuples in its descriptor.

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 a mode.

2.2.4.1. continued 1

h) A given mode is "adapted (adjusted, arrayed, united) from" another mode if the notion consisting of that other mode followed by 'operand' is a production of the notion consisting of 'adapted ('adjusted', 'arrayed', 'united') followed by the given mode followed by 'operand' {see 8.2.}.

{e.g. The mode specified by *real* is adjusted from the mode specified by *ref real*. }

i) A mode "envelops" a given mode if it either is that given mode or is united from it.

j) A given mode is "derived from" another mode if the given mode is a production of 'procedure PARAMETY delivering a LMODE', the other mode is a production of 'procedure PARAMETY delivering a RMODE', the two productions of 'PARAMETY' are the same and the production of 'LMODE' is adapted from the production of 'RMODE'.

{The mode specified by *real* is widened from the mode specified by *int*, and thus the mode specified by *proc(real) real* is derived from the mode specified by *proc(real) int*. }

k) Two modes are "related" if each is derived, adjusted or arrayed from one same mode {see 4.4.2.}.

{e.g. The pairs of modes specified by *real* and *proc real*, *int* and *[1 : n] int*, *proc union(int, char)* and *proc int* are related.}

l) The mode of a structured value "covers" a given mode if the largest mode in one of its field modes {d} is that given mode or, otherwise, covers it.

2.2.4.1. continued 2

{e.g. In the context of the declarations
struct *a* = (*aa*, *bb*) and
struct *b* = (*aa*, *ref* *bb*),
the mode specified by *a* covers those specified by *a* and *b*, whereas the mode specified by *b* covers that specified by *a* but not that specified by *ref* *b*. }

2.2.4.2. Scopes, inner and outer scopes

a) Each value has one specific "scope". At each instance, a value has, moreover, one specific "inner scope" and "outer 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 possessed by a given external object is the smallest "range" {4.1.1.e} containing a defining occurrence of an identifier (indication), if any, applied but not defined within that external object and otherwise, the program, and
that of a name is some {8.5.2.c} range.

c) The inner (outer) scope of a value possessed by an external object whose value is constrained to have one fixed scope is that scope.

d) The inner (outer) scope of a value possessed by an external object whose value is constrained to have one of a number of inner (outer) scopes, is the smallest (largest) of those inner (outer) scopes.

2.2.5. Operations

a) An operator denotes a monadic (dyadic) "operation", i.e. a routine which maps a (pair of) value(s) onto a value {7.3, 7.5, 10.2}.

b) Operations on real numbers when said to be performed "in the sense of numerical analysis" {10.2.3.g, 1, m}, are performed on real numbers which may deviate slightly from the given ones; this deviation is not defined in this Report.

2.2.6. Actions

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

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

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

2.3. Semantics

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

a) The elaboration of a program is the elaboration of the closed-statement {6.4.1.a} consisting of the same sequence of symbols.
{In this Report, the Syntax specifies the structure which a sequence of symbols has if it is a production of 'program', and under Semantics the actions performed by the computer when elaborating a program are described. Both are recursive. }

2.3. continued

b) In ALGOL 68, a specific notation for external objects is used which, together with its recursive definition, makes it possible to handle and to distinguish between arbitrarily long sequences of symbols, to distinguish between arbitrarily many different values of a given mode (except 'boolean') and to distinguish between arbitrarily many modes, which allows arbitrarily many objects to occur in the computer and which allows the elaboration of a program to involve an arbitrarily large, not necessarily finite, number of actions.
This is not meant to imply that the notation of the objects in the computer is that used in ALGOL 68 nor that it has the same possibilities. It is, on the contrary, not assumed that the computer can handle arbitrary amounts of presented information. It is not assumed that these two notations are the same or even that a one-to-one correspondence exists between them; in fact, the set of different notations of objects of a given category may be finite. It is not assumed that the number of objects and relationships that can be established is sufficient to cope with the requirements of a given program nor that the speed of the computer is sufficient to elaborate a given program within a prescribed lapse of time.

c) A model of the hypothetical computer, using an actual 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 additional restrictions are formulated defining a language whose programs are a subset of the programs of ALGOL 68, 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 "possibly intended" program. Whether a program or one of the possibly intended programs describes the process its writer in fact intended to describe is a matter which falls outside of this Report. }

3. Tokens

3.0. Syntax

3.0.1. Introduction

a)* token : basic token ; special token ; other token.
b) basic token : letter token ; denotation token ; action token ;
declaration token ; syntactic token ; sequencing token ;
reservation token ; extra token.
c) special token : quote symbol ; comment symbol.
d) other token : other character token ; other multicharacter token.
e) NOTION option : NOTION ; EMPTY.
f) chain of NOTIONs separated by SEPARATORs : NOTION ;
NOTION, SEPARATOR, chain of NOTIONs separated by SEPARATORs.
g) NOTION list : chain of NOTIONs separated by comma symbols.
h) NOTION sequence : chain of NOTIONs separated by EMPTYs.
i) NOTION pack : open symbol, NOTION, close symbol.

{Examples:
a) + ; " ; *plus* ;
b) *a* ; *0* ; + ; *int* ; *if* ; . ; *nil* ; *for* ;
c) " ; *c* ;
d) ? ; *plus* ;
e) *0* ; ;
f) *0, 1, 2* ;
g) *0* ; *0, 1, 2* ;
h) *0* ; *000* ;
i) *(1, 2, 3)* }

{For letter-tokens see 3.0.2, for denotation-tokens see 3.0.3, for action-tokens see 3.0.4, for declaration-tokens see 3.0.5, for syntactic-tokens see 3.0.6, for sequencing-tokens see 3.0.7, for reservation-tokens see 3.0.8, for extra-tokens see 3.0.9 and for other-character-tokens and other-multicharacter-tokens see 1.1.5. Step 3. }

3.0.2. Letter tokens

a) letter token : LETTER.

{Examples:
a) *a* ; π (see 1.1.4.a) }

{Letter-tokens are constituents of identifiers (4.1.1.a), field-selectors (7.1.1.i), character-denotations (5.1.4), row-of-character-denotations (5.3) and pattern-denotations (5.1.5). }

3.0.3. Denotation tokens

a) denotation token : number token ; truefalse ;yinyang ; asterisk symbol ; format symbol ; expression symbol ; parameter symbol ; flipflop ; comma symbol ; space symbol.
b) number token : digit token ; point symbol ; times ten to the power symbol.
c) digit token : DIGIT.
d) truefalse : true symbol ; false symbol.
e) yinyang : yin symbol ; yang symbol.
f) flipflop : flip symbol ; flop symbol.

{Examples:
a) *1* ; *true* ; ' ; * ; *f* ; *expr* ; : ; *1* ; , ; . ;
b) *1* ; . ; 10 ;
c) *1* ;
d) *true* ; *false* ;
e) ' ; ' ;
f) *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 expression-symbol, serve only to construct denotations. }

3.0.4. Action tokens

a) action token : operator token ; relator token ; value of symbol ;
    becomes symbol.
b) operator token : or symbol ; and symbol ; not symbol ; equals symbol ;
    differs from symbol ; is less than symbol ; is at most symbol ;
    is at least symbol ; is greater than symbol ; plusminus ;
    times symbol ; divided by symbol ; quotient symbol ; modulo symbol ;
    absolute value of symbol ; lengthen symbol ; shorten symbol ;
    round symbol ; sign symbol ; entier symbol ; odd symbol ;
    representation symbol ; real part of symbol ;
    imaginary part of symbol ; conjugate symbol ; binal symbol ;
    to the power symbol.
c) plusminus : plus symbol ; minus symbol.
d) relator token : is symbol ; is not symbol ;
    conforms to symbol ; conforms to and becomes symbol.

{**Examples:**
a) + ; ::= ; *val* ; := ;
b) ∨ ; ∧ ; ¬ ; = ; ≠ ; < ; ≤ ; ≥ ; > ; + ; × ; / ;
    ↑ ; ↑: ; *abs* ; *leng* ; *short* ; *round* ; *sign* ; *entier* ; *odd* ; *repr* ;
    re ; *im* ; *conj* ; *bin* ; ↑ ;
c) + ; - ;
d) :=: ; :≠: ; :: ; ::= }

{Operator-tokens are constituents of formulas (8.1). An operator-token may be caused to denote an operation by the elaboration of an operation-declaration (7.5).
Relator-tokens are constituents of identity-relations (8.1.1) or conformity-relations (8.1.2).
Becomes-symbols are constituents of assignations (8.8) }

3.0.5. Declaration tokens

a) declaration token : PRIMITIVE symbol ; long symbol ;
reference to symbol ; procedure symbol ; structure symbol ;
union of symbol ; local symbol ; complex symbol ; bits symbol ;
string symbol ; mode symbol ; ADIC symbol ; denotes symbol ;
operation symbol.

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

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

3.0.6. Syntactic tokens

a) syntactic token : open ; close ; elementary symbol ; parallel symbol ;
sub symbol ; bus symbol ; up to symbol ; at symbol ; if symbol ;
THELSE symbol ; fi symbol ; of symbol ; label symbol.
b) open : open symbol ; begin symbol.
c) close : close symbol ; end symbol, TAG option.

{Examples:
a) (;) ; *elem* ; *par* ; [;] ; : ; : ; *if* ; *then* ; *fi* ; *of* ; : ;
b) (; *begin* ;
c)) ; *end* ; *end zero* }

{Syntactic-tokens are constituents of phrases, in which they separate terminal productions of notions or group them together. }

3.0.7. Sequencing tokens

a) sequencing token : go on symbol ; completion symbol ; go to symbol.

{Examples:
a) ; ; . ; *go to* }

{Sequencing-tokens are constituents of phrases, in which they specify the order in which phrases are elaborated (6.1). }

3.0.8. Reservation tokens

a) reservation token : skip symbol ; nil symbol.

{Examples:
a) *skip* ; *nil* }

{Reservation-tokens are constituents of skips (6.2.1.e) and nihils (8.3.1.d). }

3.0.9. Extra tokens and comments

a) extra token : for symbol ; from symbol ; by symbol ; to symbol ; while symbol ; do symbol ; case symbol ; plus i times symbol.
b) comment : comment symbol, comment item sequence option, comment symbol.
c) comment item : basic token ; quote symbol.

{Examples:
a) *for* ; *from* ; *by* ; *to* ; *while* ; *do* ; *case* ; *i* ;
b) *c with respect to c* ;
c) *w* ; " }

{Extra-tokens and comments may occur in phrases which, by virtue of the extensions of Chapter 9, stand for phrases 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 the converse does not hold. }

3.1. Symbols

3.1.1. Representations

a) Letter tokens

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

b) Denotation tokens

symbol	representation
digit zero symbol	*0*
digit one symbol	*1*
digit two symbol	*2*
digit three symbol	*3*
digit four symbol	*4*
digit five symbol	*5*
digit six symbol	*6*
digit seven symbol	*7*
digit eight symbol	*8*
digit nine symbol	*9*
point symbol	.
times ten to the power symbol	10 *e*

3.1.1. continued 1

symbol	representation
true symbol	*true*
false symbol	*false*
yin symbol	' *6*
yang symbol	, *9*
asterisk symbol	*
format symbol	*f*
expression symbol	*expr*
parameter symbol	:
flip symbol	*1*
flop symbol	*0*
comma symbol	,
space symbol	.

c) Action tokens

symbol	representation
or symbol	∨ *or*
and symbol	∧ *and*
not symbol	¬ *not*
equals symbol	=
differs from symbol	≠ *neq*
is less than symbol	< *lss*
is at most symbol	≤ *leq*
is at least symbol	≥ *geq*
is greater than symbol	> *grt*
plus symbol	+
minus symbol	- *
times symbol	×
divided by symbol	/
quotient symbol	÷ *quotient*
modulo symbol	÷: *mod*
absolute value of symbol	*abs*

3.1.1. continued 2

symbol	representation		
lengthen symbol	*leng*		
shorten symbol	*short*		
round symbol	*round*		
sign symbol	*sign*		
entier symbol	*entier*		
odd symbol	*odd*		
representation symbol	*repr*		
real part of symbol	*re*		
imaginary part of symbol	*im*		
conjugate symbol	*conj*		
binal symbol	*bin*		
to the power symbol	↑	*power*	**:
is symbol	:=:	*is*	
is not symbol	:≠:	*is not*	
conforms to symbol	::	*may be*	
conforms to and becomes symbol	::=	*may become*	
value of symbol	*val*		
becomes symbol	:=		

d) Declaration tokens

symbol	representation
integral symbol	*int*
real symbol	*real*
boolean symbol	*bool*
character symbol	*char*
pattern symbol	*format*
long symbol	*long*
reference to symbol	*ref*
procedure symbol	*proc*
structure symbol	*struct*
union of symbol	*union*

3.1.1. continued 3

symbol	representation
local symbol	*loc*
complex symbol	*compl*
bits symbol	*bits*
string symbol	*string*
mode symbol	*mode*
monadic symbol	*monadic*
dyadic symbol	*dyadic*
denotes symbol	=
operation symbol	*op*

e) Syntactic tokens

symbol	representation	
open symbol	(	
begin symbol	*begin*	
close symbol	)	
end symbol	*end*	
elementary symbol	*elem*	
parallel symbol	*par*	
sub symbol	[	(
bus symbol	]	)
up to symbol	:	
at symbol	:	*at*
if symbol	(	*if*
then symbol	|	*then*
else symbol	|	*else*
fi symbol	)	*fi*
of symbol	*of*	
label symbol	:	

3.1.1. continued 4

f) Sequencing tokens

symbol	representation	
go on symbol	;	
completion symbol	.	*exit*
go to symbol	*go to*	*goto*

g) Reservation tokens

symbol	representation
skip symbol	*skip*
nil symbol	*nil*

h) Extra tokens

symbol	representation	
for symbol	*for*	
from symbol	*from*	
by symbol	*by*	
to symbol	*to*	
while symbol	*while*	
do symbol	*do*	
case symbol	*case*	
plus i times symbol	|	*i*

i) Special tokens

symbol	representation	
quote symbol	"	
comment symbol	*c*	*comment*

3.1.2. Remarks

a) Where more than one representation of a symbol is given, any one of them may be chosen.

{However, discretion should be exercised, since the text

(a > b then b | a fi,

though acceptable to an automaton, would be more intelligible to a human in either of the two representations

(a > b | b | a)

or

if a > b then b else a fi . }

b) The fact that the 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.a}.

c) If, in a text intended to be a program, a mark occurs which does not match one of the given representations and does not represent an other letter-token {1.1.4.a}, then it represents an other-token.

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

3.2. Semantics

A character-token {5.1.4.1.b} denotes a character {2.2.3.1.a}; an other-multicharacter-token {3.0.1.d} denotes a multiple value with more than one element, each being a character; a token {3.0.1.a} other than these denotes a multiple value with one or more elements, each being a character.

4. Identification and context conditions

4.1. Identifiers

4.1.1. Syntax

a)* identifier : MABEL identifier.
b) MABEL identifier : TAG.
c) TAG LETTER : TAG, LETTER.
d) TAG DIGIT : TAG, DIGIT.
e)* range : COERCETY serial CLAUSE ; PROCEDURE denotation.

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

{Rule b, together with 1.2.2.s 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 symbol letter b symbol.

is one such production rule. One of the production rules of the strict language arising similarly from rule c is

letter a symbol letter b symbol : letter a symbol, letter b symbol.

yielding a terminal production of 'real identifier'.
See also 7.1.1.g and 8.6 for additional insight into the function of rules c and d. }

4.1.2. Identification of identifiers

a) At a given occurrence, an identifier is either "defined" or "applied". It is defined if
   i) it occurs following a formal-declarer {7.4.1.a},
   ii) within some range, it is the first occurrence of that identifier in a constituent flexible-lower-bound or flexible-upper-bound {7.1.1.u} of that range, or
   iii) it occurs in a label {6.1.1.g};
otherwise, it is applied.

4.1.2. continued

b) If, at a given occurrence, 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, if any, contained within it, is called the home and Step 3 is taken {; otherwise, 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-expression *(bits x(101) ; abs x[2] = 0)*, 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 (4.4.1), is also a reference-to-row-of-boolean-identifier.}

{Identifiers have no inherent meaning. The defining occurrence of an identifier either is in a label (6.1.1.g) or is made to denote a value (2.2.3) by the elaboration of an identity-declaration (7.4.2).}

4.2. Indications

4.2.1. Syntax

a)* indication : MODE mode indication ; PRIORITY ADIC indication.
b) MODE mode indication : indicator ; mode standard.
c) mode standard : string symbol ;
long symbol sequence option, complex symbol ;
long symbol sequence option, bits symbol.
d)* priority indication : PRIORITY ADIC indication.
e) PRIORITY ADIC indication : ADIC indication.
f) ADIC indication : indicator ; long symbol sequence option, operator token.
g) indicator : other token.

4.2.1. continued

{Examples:
b) *primitive* ; *compl* ;
c) *string* ; *long compl* ; *bits* ;
f) *plus* ; + ; *long abs* ;
g) *primitive* ; *plus* }

4.2.2. Identification of indications

a) At a given occurrence, an indication is either defined or applied. It is defined if it occurs preceding the equals-symbol in a mode-declaration {7.2.1.a} or priority-declaration {7.3.1.a}; otherwise, it is applied.

b) If, at a given occurrence, an indication is applied, then it may identify the indication at a defining occurrence found using the steps of 4.1.2.b, with "identifier" replaced by "indication".

{Indications have no inherent meaning. The defining occurrence of an indication establishes that indication as a terminal production of 'MODE mode indication' (7.2.1.a) or 'PRIORITY ADIC indication' (7.3.1.a).}

4.3. Operators

4.3.1. Syntax

a)* operator : RMODE PRIORITY monadic operator ;
LMODE RMODE PRIORITY dyadic operator.
b) RMODE MODE PRIORITY monadic operator :
RMODE PRIORITY monadic operator.
c) RMODE PRIORITY monadic operator : PRIORITY monadic indication.
d) LMODE RMODE MODE PRIORITY dyadic operator :
LMODE RMODE PRIORITY dyadic operator.
e) LMODE RMODE PRIORITY dyadic operator : PRIORITY dyadic indication.

{Examples:
c) *abs* ; e) + }

4.3.2. Identification of operators

a) At a given occurrence, an operator is either defined or applied. It is defined if it occurs preceding the denotes-symbol in an operation-declaration {7.5.1.a}; otherwise, it is applied.

b) If, at a given occurrence, an operator is applied, then it may identify the operator at a defining occurrence found using the steps of 4.1.2.b, with Step 3 replaced by

"Step 3: If the home contains a defining occurrence of an operator which is the same priority-indication as the given occurrence, and which {in view of the identification condition (4.4.1)} could be a defining occurrence of the given occurrence, then the given occurrence of the operator identifies the operator at that defining occurrence; otherwise, Step 2 is taken."

{Operators have no inherent meaning. The defining occurrence of an operator is made to denote a routine (2.2.3.4) by the elaboration of an operation-declaration (7.5.2).

A given occurrence of an indication may be both a priority-indication and an operator. As a priority-indication, it identifies its defining occurrence. As an operator, it identifies another defining occurrence, which denotes a routine. Since the occurrence of an indicator in an operation-declaration is an application as a priority-indication and a definition (but not an application) as an operator, it follows that the set of those occurrences which identify a given operator is a subset of those occurrences which identify the same priority-indication.

In the closed-statement

```
begin real x, y(7.5) ; dyadic min = 6 ;
op real min = (real a, b) : (a > b | b | a) ;
x := y min pi / 2 end
```

the first occurrence of *min* is a defining occurrence of a priority-SIX-dyadic-indication. At the second occurrence, the priority-indication *min* is applied and identifies the first occurrence, whereas, at the same textual position, the operator *min* is defined as a real-real-real-priority-SIX-dyadic-operator and hence is also a real-real-priority-SIX-

4.3.2. continued

dyadic-operator (4.3.1.d; i.e. ignoring the mode of the value which it delivers). At the third occurrence of *min*, it is an application of a priority-indication and, as such, identifies the first occurrence, whereas, at the same textual position, *min* is also an application of an operator, and, as such, identifies the second occurrence; this makes it (in view of the identification condition, 4.4.1) a real-real-priority-SIX-dyadic-operator and hence, also because of the identification condition, a real-real-real-priority-SIX-dyadic-operator. This identification of the operator is made because

i) *min* occurs in an operation-declaration,
ii) *y* could be an adjusted-real-priority-SIX-dyadic-formulation,
iii) *pi/2* could be an adjusted-real-priority-SIX-monadic-formulation (since it is a priority-SEVEN-dyadic-formula)
iv) *min* is a real-real-priority-SIX-dyadic-operator, and
v) this combination of possibilities satisfies the identification condition.

With this identification of the operator accomplished we now know that *y* is an adjusted-real-priority-SIX-dyadic-formulation and that *pi/2* is an adjusted-real-priority-SIX-monadic-formulation. 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.}

4.4. Context conditions

{Let α be the set of all terminal productions of 'program' as described in chapter 1. Let β be the set of all programs in α which satisfy the context conditions, and let γ be the set of all programs which are successfully assigned a meaning by this Report. The inclusion relations

$$\alpha \supset \beta \supset \gamma$$

hold and are proper. The purpose of the context conditions is to select, for consideration in this Report, only those programs in β. Whether or not the set of programs "in the language" is the set α or the set β or even the set γ, is a matter for individual taste. If the choice is β, then the context conditions become syntax which is not written in the form of production rules. }

4.4.1. The identification condition

An identifier (indication, operator) at an applied occurrence, 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 its defining occurrence.

4.4.2. The mode condition

No declarer specifies an enigmatic mode {2.2.4.1.d}, a mode which envelops {2.2.4.1.i} two related modes {2.2.4.1.k}, or a mode which covers {2.2.4.1.l} itself.

4.4.3. The uniqueness condition.

No range, excluding all ranges contained within it, contains more than one defining occurrence of a given identifier or mode-indication, or of a monadic-indication (dyadic-indication, operator) as a terminal production of a given direct production of 'monadic indication' ('dyadic indication', 'operator'), and, in a given range, no mode-indication (priority-indication) occurs as a priority-indication (mode-indication); furthermore, no program contains an applied occurrence of an identifier, indication or operator which does not identify a defining occurrence.

5. Denotations

5.0.1. Syntax

a)* denotation : PLAIN denotation ; BITS denotation ; STRING denotation ;
PROCEDURE denotation.

{Examples:
a) *3.14* ; *101* ; *"algol.report"* ; *(bool a, b) : (a | b | false)* }

5.0.2. Semantics

{A denotation denotes a value; this relationship is permanent (see 2.2.2.a). }

The mode of the value denoted by a 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 mode of *"algol.report"*, which is a production of 'row of character denotation', is 'row of character'. }

5.1. Plain denotations

5.1.0.1. Syntax

a)* plain denotation : PLAIN denotation.
b) long INTREAL denotation : long symbol, INTREAL denotation.

{Examples:
a) *4096* ; *3.14* ; *true* ; *'a'* ; *f7z3df*
b) *long 4096* ; *long long 3.141592653589793* }

5.1.0.2. Semantics

A plain-denotation denotes a plain value {2.2.3.1}, but plain values denoted by different plain-denotations are not necessarily different.

5.1.1. Integral denotations

5.1.1.1. Syntax

a) integral denotation : digit zero symbol ; natural numeral.
b) natural numeral : digit FIGURE symbol, digit token 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 written as such. The value of a denotation of an integer of given length is its a priori value provided that this does not exceed the largest integer of that length {(10.1.b); otherwise, the value is undefined (2.2.3.1.b)}.

5.1.2. Real denotations

5.1.2.1. Syntax

a) real denotation : variable-point numeral ; floating-point numeral.
b) variable-point numeral : integral part option, fractional part ;
integral part, point symbol.
c) integral part : integral denotation.
d) fractional part : point symbol,
digit zero symbol sequence option, integral denotation
e) floating-point numeral : stagnant part, exponent part.
f) stagnant part : integral denotation ; variable-point numeral.
g) exponent part : times ten to the power symbol, power of ten.
h) power of ten : plusminus option, integral denotation.

5.1.2.1. continued

{Examples

a) *0.000123* ; *1.23e-4* ; b) *.123* ; *0.123* ; *123.* ;

c) *123* ; d) *.123* ; *.000123* ;

e) *1.23e-4* ; 7_{10}+*5* 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, in the sense of numerical analysis, as many times as there are digit-tokens in the fractional-part.

b) The a priori value of a variable-point-numeral is the a priori value of its fractional-part if it has no integral-part and that of its integral-part if it has no fractional-part; otherwise, it is the sum in the sense of numerical analysis of the a priori values of its integral-part and fractional-part.

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.

e) The value of a denotation of a real number of given length is its a priori value provided that this does not exceed the largest real number of that length {(2.2.3.1.b, 10.1.d); otherwise, the value is undefined}.

5.1.3. Boolean denotations

5.1.3.1. Syntax

a) boolean denotation : truefalse.

{Examples:
a) *true* ; *false* }

5.1.3.2. Semantics

The value of a true-symbol or false-symbol is true or false respectively.

5.1.4. Character denotations

5.1.4.1. Syntax

a) character denotation : yin symbol, character token, yang symbol.
b) character token : graphic ; asterisk symbol.
c) graphic : letter token ; number token ; plusminus ; open symbol ;
close symbol ; space symbol ; quote symbol ; comma symbol ;
times symbol ; divided by symbol ; equals symbol ; other character token.

{Examples:
a) 'a' ;
b) a ; * ;
c) a ; 1 ; + ; (;) ; . ; " ; , ; / ; = ; ? }

5.1.4.2. Semantics

The value of a character-denotation is the character denoted by the character-token enclosed between the yin-symbol and the yang-symbol.

5.2. Row of boolean denotations

5.2.1. Syntax

a) BITS denotation : long symbol sequence option, flipflop sequence.

{Examples:
a) *101* ; *long 101* }

5.2.2. Semantics

a) The a priori value of a flip-symbol is true and that of a flop-symbol is false.

b) The a priori value of a flipflop-sequence consists of the values of its flipflops taken in the same order and preceded by false an infinity of times.

c) The value of a row-of-boolean-denotation is the multiple value {2.2.3.3} whose elements are the last n members of the a priori value of its flipflop-sequence, where n stands for the value of *L bits width* {10.1.f}, (*L* standing for as many times *long* as there are long-symbols in the denotation) and whose descriptor has an offset 1 and one quintuple (1, n, 1, 1, 1).

{If the value of *bits width* is, say, 5, then *101* denotes the same value as that possessed by the collateral-expression *(false, false, true, false, true)*, but *101* is not a collateral-expression. }

5.3. Row of character denotations

5.3.1. Syntax

a) STRING denotation :
quote symbol, string item sequence option, quote symbol.
b) string item : basic token ; quote image ; comment symbol ; other token.
c) quote image : quote symbol, quote symbol.

{Examples:
a) "" ; *"a"* ; *"abcde"* ; *"a.+.b.""is.a.formula""""* ;
b) *a* ; "" ; *c* ; ? ;
c) "" (Since a string may not follow a string, the production of 'quote image' does not cause ambiguities.) }

5.3.2. Semantics

a) The "row value" associated with a symbol is a multiple value whose elements are characters; if the symbol is a character-token, only element is the character denoted by the character-denotation containing that character-token; otherwise, the number of elements is one or more.

b) The row value associated with a quote-image is the row value associated with the quote-symbol.

c) The value of a row-of-character-denotation is a multiple value whose elements are the elements of the row values associated with its string-items, taken in the same order, and whose descriptor has an offset 1 and one quintuple (1, n, 1, 1, 1), where n stand for the sum of the numbers of elements of the row values associated with its string-items.

{The denotation *"ab"* denotes a value which is the same as the value possessed by the collateral-expression *('a', 'b')*, but *"ab"* is not a collateral-expression. }

5.4. Routine denotations

5.4.1. Syntax

a)* routine denotation : PROCEDURE denotation.
b) procedure with a PARAMETERS delivering a MODE denotation:
formal PARAMETERS pack, virtual MODE declarer, parameter symbol, adapted MODE base.
c) procedure with a PARAMETERS denotation :
formal PARAMETERS pack, parameter symbol, basic statement.
d) VICTAL PARAMETERS and a PARAMETER :
VICTAL PARAMETERS, comma symbol, VICTAL PARAMETER.
e) formal MODE parameter : formal MODE declarer, MODE identifier.
f) procedure delivering a MODE denotation:
virtual MODE declarer, expression symbol, adapted MODE base.
g) procedure denotation : expression symbol, basic statement.

{Examples:
b) *(bool a, b) bool : (a | b | false)* ;
c) *(ref int i) : (i > 0 | i := i - 1)* ;
d) *bool a, b* ; *ref int i* ; [] *real x* ; [*1 : 10*] *real y* ; [*int m : int n*] *real z* ;
e) *bool a* ; *ref int i* ;
f) *real expr(p | x | y)* ; *expr(p | x | y)* (9.2) ;
g) *expr(n = 1966 | warsaw | zandvoort)* }

{For adapted-bases see 8.3.1.b and for basic-statements see 6.2.1.b. }

5.4.2. Semantics

A routine-denotation denotes that routine which would be obtained from it by
i) placing an open-symbol before it and a close-symbol after it;
ii) inserting a denotes-symbol followed by a skip-symbol following the last identifier in each constituent formal-parameter;
iii) deleting the constituent virtual-declarer, if any, preceding the constituent parameter-symbol or expression-symbol;
iv) replacing the parameter-symbol, if any, by a go-on-symbol, and
v) deleting the expression-symbol, if any.

5.4.2. continued

{For the mode of a routine see 5.0.2.b. A routine is a value which may be accompanied by formal-parameters or not, and when called may deliver a value or not. The process described above implies that a routine-denotation denotes a token-sequence which is the same as a closed-clause (6.4.1.a), in which the formal-parameters, if any, have been transformed into identity-declarations (7.4.1). For the elaboration of routines see 8.1.0.2 (formulas), 8.2.1.2. (unaccompanied-calls) and 8.7 (accompanied-calls). }

6. Phrases

6.0.1. Syntax

a)* phrase : COERCETY SOME PHRASE.
b)* clause : COERCETY SOME CLAUSE.
c)* expression : COERCETY SOME MODE expression.
d)* declaration : SOME declaration.
e)* statement : SOME statement.
f)* SOME phrase : COERCETY SOME PHRASE.
g)* SOME clause : COERCETY SOME CLAUSE.
h)* SOME expression : COERCETY SOME MODE expression.

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, except in the case mentioned in 7.0.2.a.

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 not defined in this Report.}

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

6.0.2. continued

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

Step 1: If an identifier (indication) which also occurs outside the given clause is defined (as a mode-indication or priority-indication) within it, then all occurrences of this identifier (indication) within the given clause are replaced by 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 indication which also occurs outside the given clause is defined as an operator within it, then all occurences of that indication which identify this defining occurrence and this defining occurrence itself are replaced by one same new indicator which does not occur elsewhere in the program and Step 3 is taken; otherwise, the protection of the given clause is complete ;

Step 3: A copy is made of the priority-declaration containing that indication which is identified by that operator; the occurrence of that indication in the copy is replaced by that new indicator; 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 phrases

6.1.1. Syntax

a) serial declaration : chain of unitary declarations separated by go on symbols.
b) COERCETY serial CLAUSE : declaration prelude option, chain of COERCETY CLAUSE trains separated by completers.
c) declaration prelude : chain of unitary declarations separated by go on symbols, go on symbol.
d) COERCETY CLAUSE train : label sequence option, statement prelude option, COERCETY unitary CLAUSE.
e) statement prelude : chain of unitary statements separated by sequencers, sequencer.
f) sequencer : go on symbol, label sequence option.
g) label : label identifier, label symbol.
h) completer : completion symbol, label.

{Examples:

a) *real x ; real y(1) ; int n = abs j ;*
b) *n := n + 1 ; real x ; int i ; x := a + 1 ;*
c) *real x ; int i ; ;*
d) *l : true ; l1 : l2 : x := a + 1; if x > 0 then goto l3 else x := 1 - x fi ; false. l3 : y := y + 1 ; true ;*
e) *x := x + 1 ; y := 1 - y ; ;*
f) *; ; ; l : ;*
g) *l : ;*
h) *. l3 : }*

{For unitary-phrases see 6.2 and Chapters 7 and 8.}

6.1.2. Semantics

a) The elaboration of a serial-declaration is initiated by initiating the elaboration of its first constituent unitary-declaration.

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

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

d) The elaboration of a serial-phrase is
   i) interrupted (halted, resumed) upon the interruption (halting, resumption) of a constitutent unitary-phrase;
   ii) terminated upon the termination of the elaboration of a constituent unitary-phrase, and its successor {6.2.2.a} is appointed the successor of the serial-phrase.

e) The elaboration of a serial-declaration is completed upon the completion of the elaboration of its last constituent unitary-declaration.

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

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

{In *y := (x := 1.2 ; 2.3)*, the value of the serial-expression *x := 1.2 ; 2.3* is the real number denoted by *2.3*. In *xx := (real r(0.1) ; r)*, the value of the serial-expression *real r(0.1) ; r* is undefined since the scope of the name denoted by *r* is the serial-expression itself. }

6.2. Unitary phrases

6.2.1. Syntax

a) unitary statement : basic statement ; MODE assignation.
b) basic statement : cohesive statement ; called cohesion ;
called NONPROC cohesion ; CLOSED statement.
c) cohesive statement : jump ; skip ; statement call ; NONPROC expression call.
d) jump : go to symbol option, label identifier.
e) skip : skip symbol.

{Examples:
a) *goto warsaw* ; *x* := *x* + *1* ;
b) *goto grenoble* ; *stop* ; *random* ;
(x := 1 ; y := 0) ; *(x := 1, y := 0)* ; *(p | x := 1 | y := 0)* ;
c) *kootwijk* ; *skip* ; *setrandom (x)* ; *det(y2, i1)* ;
d) *goto amsterdam* ; *zandvoort* ;
e) *skip* }

{For unitary-declarations see Chapter 7, and for unitary-expressions see Chapter 8.
For assignations see 8.8.1.a, for called-cohesions see 8.2.1.1.b, c, for closed-statements see 6.4.1.a, for collateral-statements see 6.3.1.b, for conditional-statements see 6.5.1.a, for expression-calls see 8.7.1.b and for statement-calls see 8.7.1.c.}

6.2.2. Semantics

a) The elaboration of a jump terminates the elaboration of the unitary-clause which is that jump, 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.

{Note that the elaboration of a jump may terminate the elaboration of other phrases (6.1.2.d, 6.3.2.a).}

6.2.2. continued

b) The elaboration of a skip involves no action.

{Skips play a role e.g. in the semantics of routine-denotations (5.4.2.ii) and accompanied-calls (8.7.2. Step 3). }

6.3. Collateral phrases

6.3.1. Syntax

a) collateral declaration : collected declaration.
b) collateral statement : collected statement.
c) COERCETY collateral row of MODE expression :
COERCETY collected MODE expression.
d) COERCETY collected PHRASE : parallel symbol option, open symbol, COERCETY unitary PHRASE, comma symbol, COERCETY unitary PHRASE list, close symbol.

{Examples:
a) *(real x, real y)* ; (and, by 9.2.b, h) *real x, y* ;
b) *(x := 0, y := 1)* ; *(x := 0, y := 1, z := 2)* ;
c) *(x, n)* ; *(1, 2.3, 4.5)* }

6.3.2. 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 {2.2.6} 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 these 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.

6.3.2. continued

b) A collateral-phrase is elaborated in the following steps:

Step 1: Its constituent unitary-phrases are elaborated collaterally {6.3.2.a} ;

Step 2: If the collateral-phrase is a collateral-expression with n constituent unitary-expressions, then a new instance of a multiple value, which is then the value of the collateral-expression, is established as follows:

If

the n values obtained in Step 1 are not of multiple mode

then

the new multiple value is composed of new instances of these n values, indexed in the same order from 1 to n, and a descriptor consisting of an offset 1 and one quintuple (1, n, 1, 1, 1) ;

otherwise,

the n values obtained in Step 1 are of multiple mode, and

if

the lower (upper) bounds of the corresponding quintuples in their descriptors are the same,

then

the new multiple value is composed of n × r elements, each of which is a new instance of an element of each of the n multiple values obtained in Step 1, the new instance of the j-th element of the value of the i-th constituent being given the index $j + (i - 1) \times r$ (where r stands for the number of elements of the multiple value of each constituent unitary-expression), and a descriptor which consists of an offset 1 and a first quintuple (1, n, r, 1, 1) followed by copies of the m quintuples of the descriptor of the multiple value of one of the constituent unitary-expressions, subjected to the following modifications:

Calling these quintuples $(l_i, u_i, d_i, s_i, t_i)$, i = 1, 2, ..., m,

i) for i = 1, 2, ..., m, the states s_i and t_i are set to 1 ;

ii) d_m is set to 1, and

iii) for i = m, m-1, ..., 2, the stride d_{i-1} is set to $(u_i - l_i + 1) \times d_i$

{; otherwise,

the value of the collateral-expression is undefined}.

6.3.2. continued 2

{The presence of a parallel-symbol makes it possible to control the progress of the elaborations of the constituent unitary-phrases by means of the synchronization routines of 10.4.}

6.4. Closed phrases

6.4.1. Syntax

a) COERCETY closed PHRASE : elementary symbol option, open,
COERCETY serial PHRASE, close.

{Examples:
a) *(real x = u) ; elem begin i := i + 1 ; j := j + 1 end increment* }

{For serial-phrases see 6.1.}

6.4.2. Semantics

a) The elaboration of a closed-phrase is that of its constituent serial-phrase.
b) The value of a closed-expression is that, if any, of its constituent serial-expression.
c) The elaboration of a closed-phrase which begins with an elementary-symbol is an elementary action {2.2.6.a}.

6.5. Conditional clauses

6.5.1. Syntax

a) COERCETY conditional CLAUSE :
if symbol, COERCETY choice CLAUSE, fi symbol.
b) COERCETY choice CLAUSE :
condition, COERCETY then CLAUSE, COERCETY else CLAUSE option.
c) condition : fitted serial boolean expression.
d) COERCETY THELSE CLAUSE : THELSE symbol, COERCETY serial CLAUSE.

6.5.1. continued

{Examples:

a) *(x > 0 | x | 0)* ("adapted" in *y* := *(x > 0 | x | 0)*) ;
if overflow then exit fi ;

b) *x > 0 | x | 0 ; overflow then exit* ;

c) *x > 0 ; overflow* ;

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

{For serial-clauses see 6.1.1.b.}

6.5.2. Semantics

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

Step 1: The condition is elaborated ;

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

Step 3: The clause following the then-symbol or else-symbol of the clause selected in Step 2, if any, is elaborated ;

Step 4: If the conditional-clause is a conditional-expression, then its value is that of the clause elaborated in Step 3, if any; otherwise, its value is undefined.

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 selected clause;

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

iii) terminated upon the termination of the elaboration of the condition or selected clause, and, if one of these appoints a successor, then this is the successor of the conditional-clause.

7. Unitary declarations

7.0.1. Syntax

a) unitary declaration : mode declaration ;
priority declaration ; identity declaration ;
operation declaration ; closed declaration ; collateral declaration.

{Examples:

a) *mode bits* = [*1 : bits width*] *bool* ;
dyadic plus = *7* ;
int m = *4096* ; *real x* ; *bool complete*(*false*) ;
proc int sgn = (*real x*) : (*x* = *0* | *1* | *sign x*) ;
op int ÷ = (*real a, b*) : (*round a* ÷ *round b*) ;
(*real x* = *u*) ; *real x, y* }

7.0.2. Semantics

a) If, during the elaboration of an expression contained within a unitary-declaration, a jump is elaborated {6.2.2.a} whose successor is a unitary-clause outside that declaration but within the smallest range containing it, then the further elaboration is undefined.

b) An identifier or indication which was caused to denote an internal object by the elaboration of a declaration ceases to denote that object upon termination or completion of the elaboration of the smallest range containing that declaration.

{For mode-declarations see 7.2, for priority-declarations see 7.3, for identity-declarations see 7.4, for operation-declarations see 7.5, for closed-declarations see 6.4 and for collateral-declarations see 6.3. The elaboration of the closed-expression

begin[*1 : (go to e ; 5)*] *int a ; e : a*[*1*] := *1 end*

is undefined, according to a. }

7.1. Declarers

7.1.1. Syntax

a)* declarer : VICTAL MODE declarer.
b) VICTAL MODE declarer :
    VICTAL MODE declarator ; MODE mode indication.
c) VICTAL PRIMITIVE declarator : PRIMITIVE symbol.
d) VICTAL long INTREAL declarator :
    long symbol, VICTAL INTREAL declarator.

{Examples:
b) *real* ; *bits* ;
c) *int* ; *real* ; *bool* ; *char* ; *format* ;
d) *long int* ; *long long real* }

e) VICTAL structured with a FIELDS declarator :
    structure symbol, FIELDS declarator pack.
f) FIELDS and a FIELD declarator :
    FIELDS declarator, comma symbol, FIELD declarator.
g) MODE named TAG declarator :
    actual MODE declarer, MODE named TAG selector.
h) MODE named TAG selector : TAG.
i)* field selector : FIELD selector.

{Examples:
e) *struct(string name, real value)* ;
f) *string name, real value* ;
g) *string name* ;
h) *name* }

j) virtual reference to MODE declarator :
    reference to symbol, virtual MODE declarer.
k) actual reference to MODE declarator :
    reference to symbol, virtual MODE declarer.
l) formal reference to NONREF declarator :
    reference to symbol, formal NONREF declarer.
m) formal reference to reference to MODE declarator :
    reference to symbol, virtual reference to MODE declarer.

7.1.1. continued

{Examples:
j) *ref* [] *real* ;
k) *ref* [] *real* ;
l) *ref*[*1 : int n*] *real* ;
m) *ref ref* [] *real* }

n) VICTAL ROWS NONROW declarator : sub symbol, VICTAL ROWS rower,
bus symbol, VICTAL NONROW declarer.
o) VICTAL row of ROWS rower :
VICTAL row of rower, comma symbol, VICTAL ROWS rower.
p) VICTAL row of rower :
VICTAL lower bound, up to symbol, VICTAL upper bound.
q) formal LOWPER bound :
flexible LOWPER bound ; strict LOWPER bound ; virtual LOWPER bound.
r) actual LOWPER bound : strict LOWPER bound ; virtual LOWPER bound.
s) virtual LOWPER bound : EMPTY.
t) strict LOWPER bound : fitted unitary integral expression.
u) flexible LOWPER bound : integral declarator symbol, integral identifier.

{Examples:
n) [*1 : m, 1 : n*] *real* ;
o) *1 : m, 1 : n* ;
p) *1 : m*
q) *int n* ; *i + j* ; ;
r) *i + j* ; ;
t) *i + j* ;
u) *int n* }

v) VICTAL procedure declarator : procedure symbol.
w) VICTAL procedure with a PARAMETERS declarator :
procedure symbol, virtual PARAMETERS pack.
x) VICTAL procedure PARAMETY delivering a MODE declarator :
VICTAL procedure PARAMETY declarator, virtual MODE declarer.
y) virtual MODE parameter : virtual MODE declarer.

{Examples:
v) *proc*
w) *proc(real, int)*
x) *proc(real, int) bool*
y) *real* }

z) VICTAL union of MODES mode declarator :
union of symbol, virtual MODES declarer pack.
aa) virtual MODES and MODE declarer :
virtual MODES declarer, comma symbol, virtual MODE declarer.

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

{A declarer is, by rule b, either a declarator or a mode-indication (4.2.1.b). A declarator may contain a declarer and thus a mode-indication, but it never begins with one.

Rule g, 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 field-selection (8.6). Concerning the occurrence of a given field-selector more than once in a declarer, see 4.4.2, 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,
struct(string name, ref cell next) and
link(ref link next, ref cell value)
may both be present in some range.

7.1.1. continued 3

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

7.1.2. Semantics

a) A given declarer specifies that mode which is obtained by deleting 'declarer' and the terminal production of the metanotion 'VICTAL' from that direct production {1.2.2.b} of the notion 'declarer' of which the given declarer is a production.

b) A given declarer is developed as follows:
Step: If it is, or contains, a mode-indication which is either an actual-declarer not preceded by a reference-to-symbol, or a formal-declarer not preceded by two reference-to-symbols, then that indication is replaced by a copy of the constituent actual-declarer of that mode-declaration {7.2.1.a} which contains the defining occurrence {4.2.2.b} of that indication, and the Step is taken again; otherwise, the development of the declarer is complete.

{A declarer is developed during the elaboration of an actual-declarer (c) or identity-declaration (7.4.2. Step 1). The exceptions concerning reference-to-symbols are made in order that the development of the actual-declarer in constructions like

struct person = (int age, ref person father)

may be finite.}

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 3 is taken; otherwise, if it now begins with a sub-symbol, then Step 4 is taken; otherwise, a new instance of an undefined value is considered, and Step 7 is taken ;
Step 3: All its constituent actual-declarers are elaborated collaterally {6.3.2.a}; the values referred to by the values {names} of these actual-

7.1.2. continued

declarers are made, in the given order, to be the fields of a new instance of a structured value, this structured value is considered, and Step 7 is taken ;

Step 4: All strict-lower-bounds and strict-upper-bounds contained in it but not in any constituent declarer of it, along with its last constituent actual-declarer {following the sub-symbol} are elaborated collaterally ;

Step 5: A descriptor {2.2.3.3} is established consisting of an offset.1 and as many quintuples as there are actual-lower-bounds (actual-upper-bounds) contained in the given declarer but not in any constituent declarer of it; if the i-th of these actual-lower-bounds (actual-upper-bounds) is 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} ;

Step 6: The descriptor is made to be the descriptor of a multiple value each of whose elements is a copy of the value referred to by the value {name} of the last constituent actual-declarer {Step 4}, and this multiple value is considered ;

Step 7: A name {2.2.3.5} different from all other names and whose mode is 'reference to' followed by the mode specified {7.1.2.a} by the actual-declarer, is created and made to refer to the considered value; this name is then the value of the given actual-declarer upon the completion, if any, of its elaboration.

7.2. Mode declarations

7.2.1. Syntax

a) mode declaration : mode symbol, MODE mode indication, equals symbol, actual MODE declarer.

{Examples:

a) *mode bits* = [*1 : bits width*] *bool* ;
struct compl = (*real re, im*) (see 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.

{Note that certain recursive mode-declarations may result in programs whose elaboration cannot be completed.}

7.3. Priority declarations

7.3.1. Syntax

a) priority declaration : ADIC symbol, priority NUMBER ADIC indication, equals symbol, NUMBER token.
b) zero token : digit zero symbol.
c) ONE token : digit one symbol.
d) TWO token : digit two symbol.
e) THREE token : digit three symbol.
f) FOUR token : digit four symbol.
g) FIVE token : digit five symbol.
h) SIX token : digit six symbol.
i) SEVEN token : digit seven symbol.
j) EIGHT token : digit eight symbol.
k) NINE token : digit nine symbol.

{Examples:
a) *monadic plus* = *7* ; *dyadic* + = *6* ; *dyadic lowest* = *0* }

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 8.1.0.2.}

7.4. Identity declarations

7.4.1. Syntax

a) identity declaration : formal MODE declarer, MODE identifier,
denotes symbol, actual MODE parameter.
b) actual MODE parameter :
adapted unitary MODE expression ; local MODE generator.

{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.)
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) : (1 - cos(x)) ;
proc real vers = (real x) : (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) : (x > 0 | x | 1)) ;
proc real q((real x) : (x > 0 | x | 1)) ;
b) *1 ; loc real* }

{For local-generators see 8.5.1.b.}

7.4.2. Semantics

An identity-declaration is elaborated in the following steps:
Step 1: Its constituent formal-declarer is developed {7.1.2.b} ;
Step 2: Its constituent actual-parameter, and all strict-lower-bounds
and strict-upper-bounds contained in the formal-declarer, as possibly
modified by Step 1, but not contained in any constituent declarer of
that formal-declarer, are elaborated collaterally {6.3.2.a} ;
Step 3: If the value of the constituent actual-parameter refers to an
element or subvalue of a multiple value {2.2.3.3} having one or more
states equal to 0, then the further elaboration is undefined ;

7.4.2. continued

Step 4: Each defining occurrence {4.1.2.a}, if any, of an identifier in a constituent flexible-lower-bound or flexible-upper-bound of the formal-declarer is made to denote a new instance of the value of the corresponding bound in the {multiple} value of the constituent actual-parameter ;

Step 5: If the value of any constituent strict-lower-bound or strict-upper-bound, or the value of any identifier {8.3.2.a} in a constituent flexible-lower-bound or flexible-upper-bound of the formal-declarer is not the same as that of the corresponding bound in the value of the constituent actual-parameter, then the further elaboration is undefined; otherwise, the constituent identifier following the constituent formal-declarer of the identity-declaration is made to denote the value of the constituent actual-parameter.

{According to Step 5, the elaboration of the declaration

[1 : 2] real x = (1.2, 3.4, 5.6)

is undefined, as is that of

[1 : int n, 1 : int n] real x = ((1.1, 1.2), (2.1, 2.2), (3.1, 3.2)).}

7.5. Operation declarations

7.5.1. Syntax

a) operation declaration : operation symbol, OPERATOR declarator, OPERATOR, denotes symbol, OPERATOR body.

b) RMODE MODE PRIORITY monadic operator declarator : virtual RMODE parameter pack, virtual MODE declarer.

c) LMODE RMODE MODE PRIORITY dyadic operator declarator : open symbol, virtual LMODE parameter, comma symbol, virtual RMODE parameter, close symbol, virtual MODE declarer.

d) RMODE MODE PRIORITY monadic operator body : adapted unitary procedure with a RMODE parameter delivering a MODE expression.

e) LMODE RMODE MODE PRIORITY dyadic operator body : adapted unitary procedure with a LMODE parameter and a RMODE parameter delivering a MODE expression.

f) * operator body : OPERATOR body.

7.5.1. continued

{Examples:

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

b) *(real) real* ;

c) *(bool, bool) bool* ;

d) *(real a) : (a < 0 | -a | a)* ;

e) *(bool a, b) : (a | b | false)* }

{For operators see 4.3.1, for virtual-parameters see 7.1.1.y, for virtual-declarers see 7.1.1 and for unitary-expressions see Chapter 8. See also routine-denotations (5.4). }

7.5.2. Semantics

An operation-declaration is elaborated in the following steps:

Step 1: Its constituent operator-body is elaborated ;

Step 2: Its constituent operator is made to denote the {routine which is the} value of the operator-body.

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 c ; {10.2.6.a}
h) *op int abs = (char a) :*
c the integral equivalent of the value of the character "a" c ;
i) *op char repr = (int a) :*
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) *dyadic* ∨ = 2, ∧ = 3, = = 4, ≠ = 4, < = 5, ≤ = 5, ≥ = 5, > = 5, + = 6,
- = 6, × = 7, / = 7, ÷ = 7, ÷: = 7, ↑ = 8 ;
b) *monadic* ¬ = 3, *leng* = 7, *short* = 7, *round* = 7, *sign* = 7,
odd = 7, + = 7, - = 7, *entier* = 7, *repr* = 7, *re* = 7, *im* = 7, *conj* = 7,
bin = 7 ;
c) *monadic L abs* = 7 ;

10.2.1. Operations on boolean operands

a) *op bool* ∨ = (*bool a, b*) : (*a | true | b*) ;
b) *op bool* ∧ = (*bool a, b*) : (*a | b | false*) ;
c) *op bool* ¬ = (*bool a*) : (*a | false | true*) ;
d) *op bool* = = (*bool a, b*) : ((*a* ∧ *b*) ∨ (¬*a* ∧ ¬*b*)) ;
e) *op bool* ≠ = (*bool a, b*) : (¬(*a = b*)) ;
f) *op int abs* = (*bool a*) : (*a | 1 | 0*) ;

10.2.2. Operations on integral operands

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

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

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

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

e) *op bool* ≥ = (*L int a, b*) : (*b* ≤ *a*) ;

f) *op bool* > = (*L int a, b*) : (*b* < *a*) ;

g) *op L int* - = (*L int a, b*) :
c the L integral value of "a" minus that of "b" c ;

h) *op L int* - = (*L int a*) : (*L* 0 - *a*) ;

i) *op L int* + = (*L int a, b*) : (*a* - - *b*) ;

j) *op L int* + = (*L int a*) : *a* ;

k) *op L int abs* = (*L int a*) : (*a* < *L0* | -*a* | *a*) ;

l) *op L int* × = (*L int a, b*) : (*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* ÷ = (*L int a, b*) :
(*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* ÷: = (*L int a, b*) : (*a* - *a* ÷ *b* × *b*) ;

o) *op L real* / = (*L int a, b*) : (*L real* (*a*) / *L real* (*b*)) ;

p) *op long L int leng* = (*L int a*) : *c the long L integral value equivalent to the L integral value of "a" c* ;

q) *op L int short* = (*long L int a*) : *c the L integral value, if it exists, equivalent to the long L integral value of "a" c* ;

r) *op L int* ↑ = (*L int a, int b*) :
(*b* ≥ *0* | *L int p*(*L1*) ; *to b do p* := *p* × *a* ; *p*) ;

s) *op bool odd* = (*L int a*) : (*a* ÷: *L2* = *L1*) ;

t) *op int sign* = (*L int a*) : (*a* > *L0* | *1* | (*a* < *L0* | *-1* | *0*)) ;

10.2.3. Operations on real operands

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

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

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

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

e) op bool ≥ = (L real a, b) : (b ≤ a) ;

f) op bool > = (L real a, b) : (b < a) ;

g) op L real - = (L real a, b) : c the L real value of "a" minus that of "b", in the sense of numerical analysis c ; {2.2.5.b}

h) op L real - = (L real a) : (L0 - a) ;

i) op L real + = (L real a, b) : (a - - b) ;

j) op L real + = (L real a) : a ;

k) op L real abs = (L real a) : (a < L0 | -a | a) ;

l) op L real × = (L real a, b) : c the L real value of "a" times that of "b", in the sense of numerical analysis c ; {2.2.5.b}

m) op L real / = (L real a, b) : c the L real value of "a" divided by that of "b", in the sense of numerical analysis c ; {2.2.5.b}

n) op long L real leng = (L real a) :
c the long L real value equivalent to the L real value of "a" c ;

o) op L real short = (long L real a) : c the L real value, if it exists, equivalent to the long L real value of "a" c ;

p) op L int round = (L real a) : c a L integral value, if one exists, equivalent to a L real value differing by not more than one-half from the L real value of "a" c ;

q) op int sign = (L real a) : (a > L0 | 1 | (a < L0 | -1 | 0)) ;

r) op L int entier = (L real a) : (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 L real P = (L real a, L int b) : (a P L real(b)) ;
b) op L real P = (L int a, L real b) : (L real(a) P b) ;
c) op bool R = (L real a, L int b) : (a R L real(b)) ;
d) op bool R = (L int a, L real b) : (L real(a) R b) ;
e) op L real ↑ = (L real a, int b) : (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 L real re = (L compl a) : re of a ;
c) op L real im = (L compl a) : im of a ;
d) op L real abs = (L compl a) : L sqrt((re a) ↑ 2 + (im a) ↑ 2) ;
e) op L compl conj = (L compl a) : L compl(re a, - im a) ;
f) op bool = = (L compl a, b) : (re a = re b ∧ im a = im b) ;
g) op bool ≠ = (L compl a, b) : (¬(a = b)) ;
h) op L compl + = (L compl a) : a ;
i) op L compl - = (L compl a) : L compl(- re a, - im a) ;
j) op L compl + = (L compl a, b) : L compl(re a + re b, im a + im b) ;
k) op L compl - = (L compl a, b) : L compl(re a - re b, im a - im b) ;
l) op L compl × = (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 / = (L compl a, b) :
     (L real d = re(b × conj b) ; L compl n = a × conj b ;
      L compl(re n / d, im n / d)) ;
n) op long L compl leng = (L compl a) : long L compl(leng re a, leng im a) ;
o) op L compl short = (long L compl a) : L compl(short re a, short im a) ;
p) op L compl P = (L compl a, L int b) : (a P L compl(b)) ;
q) op L compl P = (L compl a, L real b) : (a P L compl(b)) ;
r) op L compl P = (L int a, L compl b) : (L compl(a) P b) ;
s) op L compl P = (L real a, L compl b) : (L compl(a) P b) ;
t) op L compl ↑ = (L compl a, int b) : (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 bool = = ([1 : int n] bool a, b) :
     (bool c(true) ; for i to n do(a[i] ≠ b[i] | c := false) ; c) ;
c) op bool ≠ = ([] bool a, b) : (¬(a = b)) ;
d) op [] bool ∨ = ([1 : int n] bool a, b) : ([1 : n] bool c ;
     for i to n do c[i] := a[i] ∨ b[i] ; c) ;
e) op [] bool ∧ = ([1 : int n] bool a, b) : ([1 : n] bool c ;
     for i to n do c[i] := a[i] ∧ b[i] ; c) ;
f) op bool ≤ = ([] bool a, b) : (a ∨ b = b) ;
g) op bool ≥ = ([] bool a, b) : (b ≤ a) ;
h) op [] bool ↑ = ([1 : int n] bool a, int b) : ([1 : n] bool c(a) ;
     e : (b > 0 | b := b - 1 ; for i from 2 to n do
       c[i - 1] := c[i] ; c[n] := false ; e
       | f : (b < 0 | b := b + 1 ; for i from n by - 1 to 2 do
       c[i] := c[i - 1] ; c[1] := false ; f)) ; c) ;
i) op L int L abs = (L bits a) : (L int c(L0) ;
     for i to L bits width do c := L2 × c + abs a[i] ; c) ;
j) op L bits bin = (L int a) : if a ≥ L0 then L int aa(a); L bits c ;
     for i to L bits width do(c := c ↑ -1 ; c[1] := odd(aa ÷: L2) ;
     aa := aa ÷ L2) ; c fi ;
```

10.2.7. Operations on character operands

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

10.2.8. String mode and associated operations

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

{The operation defined in b implies that if 'a' < 'b' then
"" < "a" ; "a" < "b" ; "aa" < "ab" ; "aa" < "ba" ; "ab" < "b". }

10.3. Standard mathematical constants and functions

a) L real L pi = c π as a L real value ; see Math. of Comp. v. 16, 1962, pp. 80-99 c ;
b) proc L real L sqrt = (L real x) : c if x ≥ 0, the square root of "x" c ;
c) proc L real L exp = (L real x) : c the exponential function of "x" c ;
d) proc L real L ln = (L real x) : c the natural logarithm of "x" c ;
e) proc L real L cos = (L real x) : c the cosine of "x" c ;
f) proc L real L arccos = (L real x) :
c if abs x ≤ 1, the inverse cosine of "x", L0 ≤ L arccos(x) ≤ L pi c ;
g) proc L real L sin = (L real x) : c the sine of "x" c ;
h) proc L real L arcsin = (L real x) :
c if abs x ≤ 1, the inverse sine of "x", abs L arcsin(x) ≤ L pi/L2 c ;
i) proc L real L tan = (L real x) : c the tangent of "x" c ;
j) proc L real L arctan = (L real x) :
c the inverse tangent of "x", abs L arctan(x) ≤ L pi/L2 c ;
k) proc L real L random = expr c the next pseudo-random L real value with uniform distribution on the interval [L0, L1) c ;
l) proc L set random = (L real x) :
c causes the next call of L random to deliver the value of "x" c ;

10.4. Synchronization routines

a) proc up = (ref int i) :
elem(i := i + 1 ; c the elaboration of all phrases, if any, whose elaboration is halted, is resumed c) ;
b) proc down = (ref int i) : (do elem(if i > 0 then i := i - 1 ; l else
c if the closed-clause 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-clause, then the elaboration of that unitary-phrase is halted; otherwise, the further elaboration is undefined
c fi) ; l : skip) ;

8. Unitary expressions

8.0.1. Syntax

a) COERCETY unitary MODE expression : COERCETY MODE formulation ;
COERCETY MODE assignation ; COERCETY MODE hop.
b) COERCED MODE hop : jump ; skip.

{Examples:
a) $x + 2 \times y$; y (in $x := y$) ; $y := 3$; $i := j$ (in $x := i := j$) ; $()$;
b) *goto grenoble* ; $()$ }

{For formulations see 8.1 and for assignations see 8.8. }

8.0.2. Semantics

A jump {see also 6.2.1.d and 6.2.2.a} does not posses a value; the value of a skip {see also 6.2.1.e and 6.2.2.b} is undefined.

8.1. Formulas

8.1.0.1. Syntax

a)* formula : MODE PRIORITY ADIC formula.
b) COERCETY MODE formulation :
COERCETY MODE priority zero dyadic formulation.
c) COERCETY MODE PRIORITY dyadic formulation :
COERCETY MODE PRIORITY monadic formulation ;
COERCETY MODE PRIORITY dyadic formula.
d) MODE PRIORITY dyadic formula : MODE PRIORITY confrontation ;
adjusted LMODE PRIORITY dyadic formulation,
LMODE RMODE MODE PRIORITY dyadic operator,
adjusted RMODE PRIORITY monadic formulation.
MODE PRIORITY confrontation.
e) COERCETY MODE PRIORITY monadic formulation :
COERCETY MODE PRIORITY plus one dyadic formulation ;
COERCETY MODE PRIORITY monadic formula.
f) MODE PRIORITY monadic formula : MODE PRIORITY depression ;
RMODE MODE PRIORITY monadic operator,
adjusted RMODE PRIORITY monadic formulation.

8.1.0.1. continued

g) boolean priority FIVE confrontation : boolean identity relation ;
boolean conformity relation.
h) COERCETY MODE priority NINE plus one dyadic formulation :
COERCETY MODE primary.

{Examples:
c) *abs* n ; $a \times b$; (priority 7)
d) $a < b \vee a = b$; (priority 2)
e) $x = y$; $\neg p$; (priority 4, 3)
f) $\neg p$; (priority 3)
g) ev :=: x ; ev ::= e ; (see 11.12.r)
h) a }

{For operators see 4.3, for identity-relations see 8.1.1, for conformity-relations see 8.1.2, for depressions see 8.1.3, for adjusted-operands see 8.2 and for primaries see 8.3. }

8.1.0.2. Semantics

A formula is elaborated in the following steps:
Step 1: A copy is made of the routine which is denoted by the operator at its defining occurrence {7.5.2, 4.3.2.b} ;
Step 2: The copy obtained in Step 1, considered as a closed-expression, is protected {6.0.2.d} ;
Step 3: If the operator is monadic (dyadic), then the skip-symbol(s) (mentioned in 5.4.2.ii) in the copy is (are) replaced by the (corresponding) adjusted-operand(s) ;
Step 4: The formula is replaced by the copy as modified in Step 3 and the elaboration of the copy is initiated.

{A formulation is either a formula or another formulation, and a formula always contains an operator-symbol (or a relator-symbol or value-symbol). A coercion is activated by a formula but is passed on from one formulation to another. There are effectively twenty priorities since the monadics occupy mezzanine positions. Not all priorities are represented by the

8.1.0.2. continued

standard operators. The following table summarises the operator-tokens and their prorities as declared in the standard-declarations (10.2.0).

priority	0	1	2	3	4	5	6	7	8	9
dyadic			∨	∧	= ≠	< ≤ > ≥	+ -	× / ÷ ÷:	↑	
monadic				¬				*abs leng short round sign odd + - entier repr re im conj bin*		

Observe that since expression-calls (8.7.1.b) in effect have a priority of ten, the closed-expression

(proc int minus = (int a) : (-a) ; -1 ↑ 2 = minus(1) ↑ 2)

has the value false.

Although the syntax defines 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.* }

8.1.1. Identity relations

8.1.1.1. Syntax

a) boolean identity relation : peeled reference to MODE relative, identity relator, peeled reference to MODE relative.

b) COERCETY MODE relative : COERCETY MODE priority FIVE monadic formulation.

c)* relative : COERCETY MODE relative.

d) identity relator : is symbol ; is not symbol.

{Examples:

a) *xx :=: yy* ; *val xx :=: x or y* (see 1.3.) ;

b) *xx* ; *val xx* ; *x or y* ;

d) :=: ; :≠: }

{For monadic-formulations see 8.1.0.1.e.}

8.1.1.2. Semantics

An identity-relation is elaborated in the following steps:
Step 1: The two relatives are elaborated collaterally {6.3.2.a} ;
Step 2: If the identity-relation contains an is-symbol (is-not-symbol), then the result yielded by its elaboration is true (false) if the values {names} obtained in Step 1 are the same and false (true) otherwise.

{Assuming the assignation *xx* := *yy* := *x* (see 1.3), the value of the identity-relation *xx* :=: *yy* is false because *xx* and *yy*, though of the same mode, do not denote the same name (2.2.3.5.b). The value of the identity-relation *val xx* :=: *x or y* has a 1/2 probability of being true because the value possessed by *val xx* is the name denoted by *x*, and the routine denoted by *x or y* (see 1.3), when elaborated, yields either the name denoted by *x* or, with equal probability, the name denoted 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. Since no automatic depressing of the relatives is provided, it must be explicitly specified, if necessary, through the use of *val* or an equivalent device. Thus, *xx* :=: *x* is not an identity-relation but *val xx* :=: *x* and *(xx := xx)* :=: *x* are. On the other hand, unaccompanied procedures will be called automatically so that *x* :=: *x or y* is also an identity-relation.
Observe that the value of the formula *1 = 2* is false, whereas *1* :=: *2* is not an identity-relation, since the values of its relatives are not names. Also,

(bool a, b) : (a | b | false) :=: *(bool b, a) : (b | a | false)*

is not an identity-relation, whereas

(bool a, b) : (a | b | false) = *(bool b, a) : (b | a | false)*

is a formula, but involves an operation which is not included in the standard-declarations. }

8.1.2. Conformity relations

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

8.1.2.1. Syntax

a) boolean conformity relation :
peeled reference to LMODE relative, conformity relator, RMODE relative.
b) conformity relator :
conforms to symbol ; conforms to and becomes symbol.

{Examples:
a) *ec* :: *e* (see 11.12.q) ; *ev* ::= *e* (see 11.12.r) ;
b) :: ; ::= }

{For relatives see 8.1.1. }

8.1.2.2. Semantics

A conformity-relation is elaborated in the following steps:
Step 1: The two relatives are elaborated collaterally {6.3.2.a} ;
Step 2: If the mode of the value of the left relative is 'reference to' followed by a mode which envelops {2.2.4.1.i} the mode of the value of the right relative, then the value of the conformity-relation is true and Step 4 is taken; otherwise, Step 3 is taken ;
Step 3: If the value of the right relative refers to another value, then this other value is said to be the value of the right relative and Step 2 is taken; otherwise, the value of the conformity-relation is false and Step 4 is taken ;
Step 4: If the conformity-relator is a conforms-to-and-becomes-symbol and the value of the conformity-relation is true, then the value of the right relative is assigned {8.8.2.a} to the value of the left relative.

{Observe that if the value of the right relative is an integer and the mode of the left relative is 'reference to' followed by a mode which envelops the mode 'real' but not the mode 'integral', then the value of the conformity-relation is false. Thus, in contrast with the assignation, no automatic widening from integral to real takes place. }

8.1.3. Depressions

8.1.3.1. Syntax

a)* depression : MODE PRIORITY depression.
b)* depressend : peeled MODE PRIORITY monadic formulation.
c) MODE priority SEVEN depression : value of symbol,
    peeled reference to MODE priority SEVEN monadic formulation.

{Examples:
c) *val xx* ; *val ec* (see 11.12.i) }

8.1.3.2. Semantics

A depression is elaborated in the following steps:
Step 1: Its constituent depressend is elaborated ;
Step 2: The value referred to by the value {name} of the depressend is the value of the depression.

8.2. Operands

8.2.0.1. Syntax

a)* operand : COERCETY OPERAND.
b)* COERCED operand : COERCED OPERAND.
c) adapted OPERAND : adjusted OPERAND ; widened OPERAND ; arrayed OPERAND.
d) adjusted OPERAND : fitted OPERAND ; expressed OPERAND ; united OPERAND.
e) fitted OPERAND : OPERAND ; called OPERAND ; depressed OPERAND.

{Examples:
c) *m* ; *n* := *m* ; *x* := *n* := *m* (in [] *real x1* = (*x* := *n* := *m*)) ;
d) *x* ; *x* ; *x* (in *union*(*bool*, *proc real*) *bpr* = *x*) ;
e) *3.14* ; *random* ; *x* (in *3.14* + *random* + *x*) }

{For called-operands see 8.2.1, for expressed-operands see 8.2.2, for depressed-operands see 8.2.3, for united-operands see 8.2.4, for widened-operands see 8.2.5 and for arrayed-operands see 8.2.6.}

{The coercion process may be illustrated by considering the analysis of *random* in *random* + *x*. Supposing that [d6] stands for 'priority-SIX-dyadic' and [m6] for 'priority-SIX-monadic', then *random* + *x*, according to 10.2.3.i, 10.2.0.a and 8.1.0.1.d is a real-[d6]-formula and *random* must therefore be an adjusted-real-[d6]-formulation, which may be produced as a procedure-delivering-a-real-identifier (see 10.3.k and 7.4.1.a) as follows:

adjusted-real-[d6]-formulation,
adjusted-real-[m6]-formulation (8.1.0.1.c),
adjusted-real-[d7]-formulation (8.1.0.1.e),
adjusted-real-[m7]-formulation (8.1.0.1.c),
... ...
adjusted-real-[d10]-formulation (8.1.0.1.e),
adjusted-real-primary (8.1.0.1.h),
adjusted-real-base (8.3.1.b),
adjusted-real-cohesion (8.3.1.c),
fitted-real-cohesion (8.2.0.1.d)
called-real-cohesion (8.2.0.1.e),
fitted-procedure-delivering-a-real-cohesion (8.2.1.1.b),
procedure-delivering-a-real-cohesion (8.2.0.1.e)
procedure-delivering-a-real-identifier (8.3.1.d).

8.2.0.1. continued

A coercion is derived from the context and is passed on through the syntactic analysis until it meets an operand (formula, cohesion, assignation), where it is activated (called, expressed, depressed, united, widened, arrayed). In the above example, the coercion was activated by a cohesion resulting in an unaccompanied-call (8.2.1). The relevant semantics appears in 8.2.1.2, where it is explained that the routine denoted by *random* must be elaborated and deliver a real value as the value of the left operand of the operator +. }

8.2.1. Unaccompanied calls

8.2.1.1. Syntax

a)* unaccompanied call : called OPERAND ; called cohesion ;
stripped OPERAND.
b) called OPERAND : fitted procedure delivering a OPERAND.
c) called cohesion : fitted procedure cohesion.
d) stripped OPERAND : peeled procedure delivering a OPERAND.
e) peeled OPERAND : OPERAND ; stripped OPERAND.

{Examples:
b) *random* (in *random* < .5) ;
c) *stop* (in ; *stop* ;) ;
d) *x or y* (in *x or y* := *a*) ;
e) *x* ; *x or y* (in *x* :=: *x or y*) }

8.2.1.2. Semantics

An unaccompanied-call is elaborated in the following steps:
Step 1: The fitted-operand or peeled-operand is elaborated and a copy is made of {the routine which is} its value ;
Step 2: The unaccompanied-call 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 unaccompanied-call before the elaboration of a successor is initiated.
{See also 8.7.2, accompanied-calls. }

8.2.2. Expressed operands

8.2.2.1. Syntax

a) expressed procedure delivering a OPERAND : expressible OPERAND.
b) expressible OPERAND : OPERAND ; expressed OPERAND ; depressed OPERAND.
c) expressed procedure cohesion : cohesive statement.

8.2.2.1. continued

{Examples:
a) *2 × random - 1* (in *proc real r1(2 × random - 1)*) ;
c) *zandvoort* (in *proc go to = zandvoort*) }

8.2.2.2. Semantics

An expressed-operand is elaborated in the following steps:
Step 1: A copy is made of the expressible-operand or cohesive-statement {itself, not its value} ;
Step 2: That routine {5.4.2} which is obtained from the copy by placing an open-symbol before it and a close-symbol after it is the value of the expressed-operand.

{If *e1*, *e2* and *e3* are label-identifiers, then the reader might recognise the effect of the declaration
[1:3] proc switch = (e1, e2, e3)
and the unitary-statement
switch[i] ;
however, the declaration
[1:3] proc switch((e1, e2, e3))
is perhaps more powerful, since the assignation
switch[2] := e1
is possible.
The elaboration of *expr(p | x | -x)* yields the routine *((p | x | -x))*, whereas that of the expressed-operand *(p | x | -x)* yields either *(x)* or *(-x)*, depending on the value of *p*. Similarly, the elaboration of *expr(x := x + 1 ; y)* yields the routine *((x := x + 1 ; y))*, whereas that of the expressed-operand *(x := x + 1 ; y)* yields, apart from a change in the value of *x*, the routine *(y)*. On the other hand, if *C* stands for a cohesive-statement (6.2.1.c) or cohesion (8.3.1.c), then the elaboration of *expr C* and that of the expressed-operand *C* both yield the routine *(C)*. }

8.2.3. Depressed operands

{"I ca'n't go no lower", said the Hatter, "I'm on the floor as it is". Alice's Adventures in Wonderland, Lewis Carroll.}

8.2.3.1. Syntax

a) depressed OPERAND : fitted reference to OPERAND.

{Example:
a) *x* (in *x* ↑ *2*) }

8.2.3.2. Semantics

A depressed-operand is elaborated in the following steps:
Step 1: The fitted-operand is elaborated ;
Step 2: The value referred to by the name yielded in Step 1 is the value of the depressed-operand.

8.2.4. United operands

8.2.4.1. Syntax

a) united union of MODE and RMODES mode FORM : adjusted MODE FORM.
b) united union of LMODES and MODE mode FORM : adjusted MODE FORM.
c) united union of LMODES and MODE and RMODES mode FORM : adjusted MODE FORM.

{Examples:
a) *one* (in *f* + *one*, see 11.12.ba) ;
b) *f* (in *f* + *one*, ibid.) ;
c) *call (fdash, g)* (in 11.12.af) }

{In a range containing
union ib = (int, bool), rb = (real, bool) ;
union rib = (real, ib) ;
as declarations,
ib ib1(1), ib2(true) ; rb rb(true) ;
rib rib1(1), rib2(ib2), rib3(1.5), rib4(p | 1 | true) ;
are initialised declarations, but
rib rib5(rb)
is not. }

8.2.5. Widened operands

8.2.5.1. Syntax

a) widened LONG real FORM : fitted LONG integral FORM.
b) widened structured with a REAL named letter r symbol letter e symbol
and a REAL named letter i symbol letter m symbol FORM :
widenable REAL FORM.
c) widenable REAL FORM : fitted REAL FORM ; widened REAL FORM.

{Examples
a, b) *1* (in *compl(1)*) }

8.2.5.2. Semantics

A widened-operand is elaborated in the following steps:
Step 1: The fitted-operand or widenable-operand is elaborated ;
Step 2: If the value yielded by Step 1 is an integer, then the elaboration of the widened-operand yields that real number which is equivalent to that integer {2.2.3.1.d} and that real number is the value of the widened-operand; otherwise, it yields that structured {complex (10.2.5)} value composed of two fields, whose field-selectors are letter-r-symbol-letter-e-symbol and letter-i-symbol-letter-m-symbol, whose modes are the same as that of the value yielded in Step 1 and which are that value and zero respectively, and that structured value is the value of the widened-operand.

8.2.6. Arrayed operands

8.2.6.1. Syntax

a) arrayed REFETY row of MODE assignation : adapted REFETY MODE assignation.
b) arrayed REFETY row of MODE PRIORITY ADIC formula :
adapted REFETY MODE PRIORITY ADIC formula.
c) arrayed REFETY row of MODE cohesion : adapted REFETY MODE cohesion option.

{Examples:
a) *x := 3.14* (in [*1 : int n*] *real a = x := 3.14*) ;
b) *x + y* (in [*1 : int n*] *real a = x + y*) ;
c) *; 1.2 ; (3.4, 5.6)*
(in [*1 : int m, 1 : int n*] *real x1 = case i of(, 1.2, (3.4, 5.6))*)}

8.2.6.2. Semantics

An arrayed-operand is elaborated in the following steps:

Step 1: If it is an adapted-operand, then this adapted-operand is elaborated, 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 called the considered array, and Step 6 is taken ;

Step 3: If the value of the adapted-operand is a name, then the value referred to by this name, and, otherwise, the value itself of the adapted-operand is called the considered value; 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 a descriptor consisting of an offset 1 and one quintuple (1, 1, 1, 1, 1) is called the considered array, and Step 6 is taken ;

Step 5: A new instance of a multiple value, called the considered array, is established, 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 ;

Step 6: If the arrayed-operand is a terminal production of a notion beginning with 'arrayed reference to', then the value of the arrayed-operand is the name which refers to the considered array, and, otherwise, is the considered array itself.

8.3. Primaries

8.3.1. Syntax

a) COERCETY MODE primary : COERCETY MODE base ; COERCETY MODE nihil.
b) COERCETY MODE base : COERCETY CLOSED MODE expression ;
COERCETY MODE cohesion.
c) MODE cohesion : MODE denotation ; MODE identifier ; MODE slice ;
nonlocal MODE generator ; MODE named TAG selection ; MODE expression call.
d) COERCED reference to MODE nihil : nil symbol.

{Examples:
a) *re of z* ; *nil* ;
b) *(a | b | false)* ; *sin(b - a)* ;
c) *true* ; *x* ; *x2[i, j]* ; *compl(1, 0)* ; *father of algol* ; *sin(b - a)* ;
d) *nil* }

{For collateral-expressions see 6.3.1.c, for closed-expressions see 6.4, for conditional-expressions see 6.5, for denotations see 5, for identifiers see 4.1, for slices see 8.4, for generators see 8.5, for field-selections see 8.6 and for expressions-calls see 8.7. }

8.3.2. Semantics

a) The value of an identifier is the value, if any, denoted by it at its defining occurrence {4.1.2, 7.4.2. Step 5}.

{The identifier *pi* as declared in the standard declaration 10.3.a, is a real-identifier (and not a reference-to-real-identifier). The value it denotes cannot be changed by assignment. In fact, in this context, *pi := 3* is not a production of 'assignation' (8.8.1.a). Similarly, the identifier *sin* as declared in 10.3.g is a procedure-with-a-real-parameter-delivering-a-real-identifier (5.4.1.b) and *sin := (real) : (x - x ↑ 3/6)* is also not an assignation. The initialised declaration *real ppi(pi)* creates a name denoted by the reference-to-real-identifier *ppi*, which name refers to the value of *pi*; moreover, another value may be assigned to that name. }

8.3.2. continued

b) {"but a grin without a cat! It's the most
curious thing I ever saw in all my life!",
Alice's Adventures in Wonderland,
Lewis Carroll.}

The value of a nihil is nil {, a name which does not refer to any value (2.2.3.5.a)}; its elaboration involves no action.

8.4. Slices

8.4.1. Syntax

a) REFETY ROWSETY ROWWSETY NONROW slice : REFETY ROWS ROWSETY NONROW whole, sub symbol, ROWS leaving ROWWSETY indexer, bus symbol.
b) NONREF whole : NONREF base ; called NONREF base.
c) reference to NONREF whole : fitted reference to NONREF base.
d) row of ROWS leaving row of ROWSETY indexer :
trimmer option, comma symbol, ROWS leaving ROWSETY indexer ;
subscript, comma symbol, ROWS leaving row of ROWSETY indexer.
e) row of ROWS leaving EMPTY indexer :
subscript, comma symbol, ROWS leaving EMPTY indexer.
f) row of leaving row of indexer : trimmer option.
g) row of leaving EMPTY indexer : subscript.
h) trimmer : actual lower bound, up to symbol,
actual upper bound, new lower part option.
i) new lower part : at symbol, new lower bound.
j) new lower bound : fitted unitary integral expression.
k) subscript : fitted unitary integral expression.
l)* trimscript : trimmer option ; subscript.

{Examples:
a) *x1[i]* ; *x2[i, j]* ; *x2[i]* ; *x1[2:n:1]* ;
b) *(1, 2, 3)* (in *(1, 2, 3)[i]*) ;
c) *x1* ; *x2* ;
d) *2:n:1, j* ; *i, 2:n:1* ;
e) *i, j* ;
f) *2:n:1* ;
g) *i* }

8.4.1. continued

{For bases see 8.3.1.b and for unitary-expressions see Chapter 8. }
{In rule a, 'ROWS' reflects the number of trimscripts in the slice, 'ROWWSETY' the number of these which are trimmer-options and 'ROWSETY' the number of 'row of' not involved in the indexer. In the slices *x2[i, j]*, *x2[i, 2:]*, *x2[i]*, these numbers are (2,0,0), (2,1,0) and (1,0,1) respectively. Because of rules h and 7.1.1.r, s, *2:3:7* ; *2:n* ; *2:* ; *:5* ; *::2* are trimmers, while rules d and f allow trimmers to be omitted. }

8.4.2. Semantics

A slice is elaborated in the following steps:

Step 1: The whole, and all subscripts, strict-lower-bounds, strict-upper-bounds and new-lower-bounds contained in the indexer are elaborated collaterally {6.3.2.a} ;

Step 2: That multiple value which is, or is referred to by, the value of the whole, is called the "considered array", 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 called the "considered trimscript", 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 $u_i \leq k \leq l_i$, 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" and "l'" are determined from the considered trimscript {trimmer option} as follows:

if the considered trimscript 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 then l' is its value, and otherwise l' is l ;

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

8.4.2. continued

Step 6: If the considered trimscript is followed by a comma-symbol, then the trimscript following that comma-symbol is called the considered trimscript, 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 array which it describes, is called the considered value; otherwise, that element of the considered array whose index is equal to the offset in the copy is called the considered value ;

Step 8: If the value of the whole 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 the considered value itself.

{A trimmer restricts the possible values of a subscript and changes its notation: first, the value of the subscript is restricted to run from the value of the strict-lower-bound up to that of the strict-upper-bound, both given in the old notation; next, all remaining values of that subscript are changed by adding the same amount to each of them, such that the lowest value then equals the value of the new-lower-bound. Thus, the assignations

y1[1 : n - 1] := x1[2 : n : 1] ; y1[n] := x1[1] ; x1 := y1

effect a cyclic permutation of the elements of *x1*. }

8.5. Generators

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

8.5.1. Syntax

a)* generator : local MODE generator ; nonlocal MODE generator.
b) local MODE generator : local symbol, nonlocal MODE generator.
c) nonlocal reference to MODE generator :
  actual MODE declarer, MODE initialisation option.
d) MODE initialisation :
  adapted CLOSED MODE expression ; MODE structure pack.
e) structured with a FIELDS and a FIELD structure :
  structured with a FIELDS structure, comma symbol,
  structured with a FIELD structure.
f) structured with a MODE named TAG structure :
  adapted unitary MODE expression ; MODE structure pack.

{Examples:
b) *loc*[*1 : 3*] *real* *(1.2, 3.4, 5.6)* ;
c) *person* ; *compl*(*1, 0*) ; *compl*(*z*) ; *string*("abs") ;
(and in the context of
struct nest = (*int a*, *struct*(*real b*, *bool c*) *d*))
nest(*1*, (*2.3*, *true*)) ;
d) (*z*) ; (*1, 0*) ;
e) *1, 0* ;
f) *1* ; (*2.3*, *true*) }

8.5.2. Semantics

a) A given structure is elaborated in the following steps:
Step 1: All constituent expressions and structures of the given structure are elaborated collaterally {6.3.2.a} ;
Step 2: 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}, the value of the structure.

8.5.2. continued

b) A generator is elaborated in the following steps:
Step 1: Its actual-declarer {7.1.2.c} and initialisation, if not empty, are elaborated collaterally, the elaboration of a structure-pack being that of its constituent structure ;
Step 2: If the initialisation is not empty, then its value is assigned {8.8.2.a} to the result {name} of the elaboration of the actual-declarer
Step 3: The value of the given generator, upon its completed elaboration, is the value {name} of the actual-declarer.

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

{Extension 9.2.a allows

ref real x = loc real

to be written

real x .

The closed-expression

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

possesses the value true, but the closed-expression

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

possesses an undefined value since the assignation *xx := x* in this latter case violates the condition on scopes (8.8.2.a. Step 1). The closed-expression

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

however, has the value true. }

{Though the value of the offset in the descriptor of a multiple value is always initially 1, this may be changed by the action of a trimmer (see 8.4.2. Step 5).

8.5.2. continued 2

The generator

[-2:3, 1: , 0:4] real

would result in the name of a multiple value, with undefined elements, whose descriptor quintuples have the values

i	l_i	u_i	d_i	s_i	t_i
1	-2	3	?	1	1
2	1	?	?	1	0
3	0	4	?	1	1

The fact that $t_2 = 0$ means that the second upper bound is virtual and its value in the descriptor may be changed by assignment (8.8.2.a). }

8.6. Field selections

8.6.1. Syntax

a)* field selection : FIELD selection.
b) REFETY FIELD selection :
FIELD selector, of symbol, REFETY selectend with a FIELD.
c) REFETY selectend with a FIELD :
REFETY structured with a FIELD whole ;
REFETY structured with a LFIELDS and a FIELD whole ;
REFETY structured with a FIELD and a RFIELDS whole ;
REFETY structured with a LFIELDS and a FIELD and a RFIELDS whole.

{Examples: The following examples are assumed in a range with the declarations

struct language = (int age, ref language father) ;
language algol(9, language(13, nil)) ;
language pl1 = language(3, algol) ;

b) *age of pl1 ; father of algol ;*
c) *algol ; pl1* }

{Rule c ensures that the value of the whole has a field selected by the field-selector in the field-selection (see 7.1.1.e, f, g, h, and the remarks below 7.1.1. and 8.6.2). The use of an identifier which looks like a field-

8.6.1. continued

selector in the same range creates no ambiguity. Thus *age of algol* := *age* is a (possible confusing to the human) assignation if the second occurrence of *age* is also an adapted-unitary-integral-expression. }

8.6.2. Semantics

A field-selection is elaborated in the following steps
Step 1: Its constituent whole is elaborated, and the structured value which is, or is referred to by, the value of that whole is considered ;
Step 2: If the value of the whole is a name, then the value of the field-selection is a new instance of the name which refers to that field of the considered structured value selected by the constituent field-selector; otherwise, it is a new instance of the value which is that field itself.

{In the examples of 8.6.1, *age of algol* is a reference-to-integral-named-[age]-selection, and, by 8.3.1.a, b, c, a reference-to-integral-primary, but *age of pl1* is an integral-named-[age]-selection and an integral-primary. (Certain pieces of text within a notion have a prolixity out of proportion to the information they convey. Thus [age] stands for 'letter-a-symbol-letter-g-symbol-letter-e-symbol' and [language] is likewise short for 'structured-with-a-integral-named-[age]-and-a-reference-to-[language]-named-[father]'. That certain notions have infinite length is clear; however, the computer can recognise them without full examination (see 7.1.2.b.).)
It follows that *age of algol* may appear as a destination (8.8.1.b) in an assignation but *age of pl1* may not. Similarly, *algol* is a reference-to-[language]-primary but *pl1* is a [language]-primary and no assignment may be made to *pl1*.
The selection *father of pl1*, however, is a reference-to-[language]-named-[father]-selection, and thus a reference-to-[language]-primary whose value is the name denoted 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 denoted by *pl1*, but there may well be a change in the [language] referred to by that name.

8.6.2. continued

By similar reasoning and because the operators *re* and *im* denote 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. }

8.7. Accompanied calls

8.7.1. Syntax

a)* accompanied call : CLAUSE call.
b) MODE expression call :
fitted procedure with a PARAMETERS delivering a MODE base,
actual PARAMETERS pack.
c) statement call : fitted procedure with a PARAMETERS base,
actual PARAMETERS pack.

{Examples:
b) *samelson(m, (int j) : x1[j])* (in the scope of
proc real samelson = (int n, proc(int) real f) :
begin long real s(long 0) ;
for i to n do s := s + (leng f(i)) ↑ 2 ;
short long sqrt(s) end) ;
c) *set random(x)* ; (see 10.3.1.) }

{For actual-parameters see 7.4.1.b and for fitted-bases see 8.3.1.b. See also unaccomapnied-calls, 8.2.1. }

8.7.2. Semantics

An accompanied-call is elaborated in the following steps:

Step 1: The fitted-base is elaborated and a copy is made of {the routine which is} its value ;

Step 2: The copy obtained in Step 1, considered as a closed-clause is protected {6.0.2.d} ;

Step 3: The copy obtained from Step 2 is modified by replacing the skip-symbols (mentioned in 5.4.2.ii) in the textual order by the actual-parameters taken in the same order ;

Step 4: The accompanied-call is replaced by the copy as modified in Step 3, and the elaboration of the copy is initiated; if this elaboration is completed or terminated, then the copy is replaced by the accompanied-call before the elaboration of a successor is initiated.

{The expression-call *samelson(m, (int j) : x1[j])* as given in the examples of 8.7.1, is elaborated by first considering (Step 1) the closed-expression

```
((int n = skip, proc(int) real f = skip) ;
 begin long real s(long 0) ;
 for i to n do s := s + (leng f(i)) ↑ 2 ;
 short long sqrt(s) end).
```

Supposing that *n*, *f*, *s* and *i* are not used elsewhere, this closed-expression is protected (Step 2) without further alteration. The actual-parameters are now inserted (Step 3) yielding the closed-expression

```
((int n = m, proc(int) real f = (int j) : x1[j]) ;
 begin long real s(long 0) ;
 for i to n do s := s + (leng f(i)) ↑ 2 ;
 short long sqrt(s) end) ,
```

and this closed-expression is elaborated (Step 4). Note that, for the duration of this elaboration, *n* denotes the same integer as that referred to by the name possessed by *m*, and *f* denotes the same routine as that denoted by the routine-denotation *(int j) : x1[j]*. During the elaboration of this and its inner nested closed-expressions (9.5.a), the elaboration of *f(i)* itself involves the elaboration of the closed-expresssion *((int j = i) ; x1[j])*, and, within this inner closed-expression, *j* denotes the same integer as that referred to by the name possessed by *i*. }

8.8. Assignations

8.8.1. Syntax

a) MODE assignation :
reference to MODE destination, becomes symbol, MODE source.
b) reference to MODE destination : peeled reference to MODE base.
c) MODE source : adapted unitary MODE expression.

{Examples:
a) *x* := *0* ; *x* := *y* ; *x* := *random* ; *xx* := *x* ; *val xx* := *1.2* ;
x1[i] := *y1[i]* := *(i = j | 1 | 0)* ; *(random < .5 | x | y)* := *1* ;
x or y := *3.4* (see 1.3.) }

{For peeled-bases see 8.3.1.b and for adapted-unitary-expressions see 8.0.1.a.}

8.8.2. Semantics

a) A value is assigned to a name in the following steps:
Step 1: If the given value does not refer to an element or subvalue of a multiple value having one or more states equal to zero {2.2.3.3.b}, and if the outer scope of the given name is not larger than the inner scope of the given value {2.2.4.2.c, d} and if the given name is not nil, then Step 2 is taken; {otherwise, the further elaboration is undefined ;}
Step 2: If the value (called the "target value") referred to by the given name is a multiple value or a structured value, then Step 3 is taken; otherwise, the target value is superseded {2.2.3.5.g} by a new instance of the given value and the assignment is complete ;
Step 3: If the target value is a structured value, then Step 5 is taken; otherwise, applying the notation of 2.2.3.3.b to the descriptor of the target value, 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 target value, as modified above, is

made to be the descriptor of a new instance of a multiple value, which is then called the target value ;

Step 4: If for all i, i = 1, ..., n, the bound l_i (u_i) in the descriptor of the target 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 element (field) of the target value is superseded by a new instance of the value of the corresponding element (field) of the given value and the assignment is complete. {The order in which these elements (fields) are superseded is undefined.}

b) An assignation is elaborated in the following steps:

Step 1: Its constituent source and destination are elaborated collaterally {6.3.2.a} ;

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

Step 3: The value of the assignation is the value of the source.

{Observe that *(x, y) := (1.2, 3.4)* is not an assignation, since *(x, y)* is not a destination; indeed, the mode of the value of a collateral-expression (6.3.1.c) does not begin with 'reference to' but with 'row of'.}

9. continued

X for one or more virtual-parameters {7.1.1.y} separated by comma-symbols,
Y for one or more formal-parameters {5.4.1.e} separated by comma-symbols, and
Z for a formal-declarer {7.1.1.b} all of whose formal-lower-bounds and formal-upper-bounds {7.1.1.q} are empty.

9.1. Comments

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 written
(m > n | m c the larger of the two c | n). }

9.2. Contracted declarations

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 written *real x* and *ref bool p = loc bool(true)* may be written *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 written *struct compl = (real re, im).* }

c) If a given unitary-declaration (field-declarator 7.1.1.g') and another unitary-declaration (field-declarator) following a comma-symbol following the given unitary-declaration (field-declarator) both begin with an occurrence of the mode-symbol, of the structure-symbol, of the union-of-symbol, of the monadic-symbol, of the dyadic-symbol, of the operation-symbol, or of one same declarer, then the second of these occurrences may be omitted.
{e.g. *real x, real y(1.2)* may be written *real x, y(1.2)* ; also *real x, real y = 1.2* may be written *real x, y = 1.2*, but the alert programmer will not wish to confuse himself by doing so (since *x* is a reference-to-real-identifier and *y* a real-identifier.).

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 requirements, by another sequence of symbols.

b) No extension may be performed within a comment {3.0.9.b} or a row-of-character-denotation {5.3}.

c) Some extensions are described in the representation language, except that

A stands for a unitary-expression {Chapter 8},
B for a unitary-expression,
C for a unitary-clause {6.2.1.a, 8},
D for the standard-declarations {2.1.b, 10} if the extension is performed outside the standard-declarations and otherwise for the empty sequence of symbols,
E for a serial-expression {6.1.1.b},
F for a unitary-expression,
G for one or more unitary-clauses separated by comma-symbols,
H for a declarer {7.1},
I for an identifier {4.1},
J for an identifier,
K for an identifier,
L for an identifier,
L for zero or more long-symbols,
M for an identifier,
N for an indication {4.2},
O for zero or one identifiers,
P for an adapted-base {8.3.1.b},
Q for a choice-clause {6.5.1.b},
R for a routine-denotation {5.4},
S for a unitary-statement {6.2.1.a},
T for a unitary-expression,
U for zero or one virtual-declarers {7.1.1.b},
V for a virtual-declarer,
W for a unitary-expression,

Note also that *mode b = bool, mode r = real* may be written *mode b = bool, r = real*, etc. }

d) If a collateral-declaration {6.3.1.a} does not begin with a parallel-symbol, is not a constituent unitary-declaration of another collateral-declaration, none of its constituent unitary-declarations is a collateral-declaration, and only its first constituent unitary-declaration {after application of 9.3.c} begins with an occurrence of a mode-symbol, structure-symbol, union-of-symbol, monadic-symbol, dyadic-symbol, operation-symbol or declarer, then its first open-symbol and last close-symbol may be omitted.

{e.g. *(real x, y, z)* may be written *real x, y, z*. }

e) *proc(X) UI = R* may be replaced by *proc UI = R*.

f) *op(X) UN = R* may be replaced by *op UN = R*.

g) *proc(X) UO(R)* may be replaced by *proc UO(R)*.

{i.e., the virtual-parameters-pack may be omitted from the constituent formal-declarer of an identity-declaration (7.4) if a routine-denotation is present, since its constituent formal-parameters display all the information given by the virtual-parameters.

e.g., the declaration *proc(real, real) real min = (real a, b) : (a > b | b | a)* may be written *proc real min = (real a, b) : (a > b | b | a)*. }

h) An actual-parameter {7.4.1.b} of the form *(Y) V : P* may be replaced *(Y) : P* ; one of the form *V expr P* may be replaced by *expr P*.

i) *op UN = (Y) V : P* may be replaced by *op UN = (Y) : P*.

{i.e., the constituent virtual-declarer which precedes the constituent parameter-symbol or expression-symbol of a routine-denotation (5.4) may be omitted if that routine-denotation occurs as an actual-parameter or operator-body (7.5.1.f.).

e.g., the declaration *proc real f = real expr(x := x + 1 ; a + 3)* may be written *proc real f = expr(x := x + 1 ; a + 3)*. }

9.2. continued 2

j) *proc* *UO((Y) V : P)* may be replaced by *proc* *UO((Y) :P)* and
proc *UO(V expr P)* may be replaced by *proc* *UO(expr P)*.

{i.e., the virtual-declarer may also be omitted if the routine-denotation, enclosed between an open-symbol and a close-symbol, forms the initialisation (8.5.1.d) of a generator (8.5).

e.g., the generator
proc real(real expr(x := x + 1 ; a + 3)) may be written
proc real(expr(x := x + 1 ; a + 3)). }

k) [:] may be replaced by [],
[:, by [,,
,:] by ,] and
,:, by ,,.

{e.g., the declarer *ref* [:] *real* may be written *ref* [] *real* and *ref* [:,:,:] *real* may be written *ref* [,,] *real*. }

9.3. Repetitive statements

a) *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) ; goto M)
fi
end ,
where *J*, *K*, *L* and *M* do not occur in *W* or *S*, 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) *begin(int J(F), int K = B) ;*
M : (int I = J ; (W | S ; (DJ := J + K) ; goto M))
end ,
where *J*, *K* and *M* do not occur in *W* or *S*, 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*.

9.3. continued

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 written
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) *(int I = A ; if DI = 1 then C fi)* may be replaced by *case A of (C)*

b) *(int I = A ; if DI = 1 then C*
else case(DI - 1) of (G) fi) may be replaced by *case A of (C, G)*.
{Examples of the use of such "case" clauses are given in 11.12.w, ap. }

c) *else if Q fi fi* may be replaced by *, Q fi*.
{e.g., *if p then princeton else if q then grenoble else zandvoort fi fi*
may be written
if p then princeton,
if q then grenoble else zandvoort fi. }

9.5. Complex values

val(L real I = A, J = B ; (DL compl(I, J)))
may be replaced by *(A | B)*.

11. Examples

11.1. Complex square root

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

a) *proc compl compsqrt = (compl z) : 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) | (ip ⊥ (y ≥ 0 | rp | - rp)))*
f) *end compsqrt*

[complex]-expression-calls {8.7.1.b} using *compsqrt* :

g) *compsqrt(w)*
h) *compsqrt(-3.14)*
i) *compsqrt(-1)*

11.2. Innerproduct1

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

```
a) proc real innerproduct1 = (int n, proc(int) real x, y) :
   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 := s + leng x(i) × leng y(i) ;
d) short s
e) end innerproduct1
```

Real-expression-calls {8.7.1.b} using *innerproduct1*:

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

11.3. Innerproduct2

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

```
a) proc real innerproduct2 = (ref[1 : int n] real a, b) :
     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 := s + leng a[i] × leng b[i] ;
d) short s
e) end innerproduct2
```

Real-expression-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-a-reference-to-integral-parameter-and-a-integral-parameter-and-a-procedure-delivering-a-real-parameter-and-a-procedure-delivering-a-real-parameter-delivering-a-real-identifier:

```
a) proc real innerproduct3 = (ref int i, int n, proc real xi, yi) :
       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 := s + leng xi × leng yi) ;
d) short s
e) end innerproduct3
```

A real-expression-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-a-reference-to-row-of-row-of-real-parameter-and-a-reference-to-real-parameter-and-a-reference-to-integral-parameter-and-a-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 := 0 ;
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
```

A statement-call {8.7.1.c} using *absmax*:

```
g) absmax(x2, x, i, j)
```

11.6. Euler summation

```
a) proc real euler = (proc(int) real f, real eps, int tim) :
     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 := sum + (ds := (abs mn < abs m[n] ∧ n < 16 |
i)                               n := n + 1 ; m[n] := mn ; mn/2 | mn))
j)         end ;
k)       sum
l) end euler
```

An expression-call using *euler*:

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

11.7. The norm of a vector

```
a) proc real norm = (ref[1 : int n] real a) :
     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 := s + (leng a[k]) ↑ 2 ;
d)    short long sqrt(s))
```

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

11.8. Determinant of a matrix

```
a) proc real det = (ref[1 : int n, 1 : int n] real a,
b)                                 ref[1 : int n] int p) :
     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 := aik - innerproduct2(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[: k1, j]))/pivot fi ;
t)           if pk ≠ k then apkj := -r fi
u)           end for j ;
v)       d := pivot × d
w)       end for k ;
x)     d
y)     end det
```

An expression-call using *det*:

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

11.9. Greatest common divisor

An example of a recursive procedure:

```
a) proc int gcd = (int a, b) :
     c the greatest common divisor of two integers c
b)     (b = 0 | 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 real / = ([1 : int n] real a, b) :
     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 :: 1]/b[2 :: 1]))
```

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 (0), one (1);
i) op bool = = (form a, ref const b) :
     (ref const ec ; (ec ::= a | val ec  :=: b | false)) ;
j) op form + = (form a, b) :
     (a = zero | b | (b = zero | a | triple(a, plus, b)));
k) op form - = (form a, b) : (b = zero | a | triple(a, minus, b));
l) op form × = (form a, b) :
     (a = zero ∨ b = zero | zero | (a = one | b | (b = one | a |
                                        triple(a, times, b))));
m) op form / = (form a, b) :
     (a = zero ∧ ¬ b = zero | zero | (b = one | a | triple(a, by, b)));
n) op form ↑ = (form a, ref const b) :
     (a = one ∨ b :=:  zero | one | (b :=:  one | a | triple(a, to, b)));
o) proc form derivative of = (form e, c with respect to c ref var x) :
p) begin ref const ec ; ref var ev ; ref triple et ; ref call ef ;
q) if ec :: e then zero,
r)    ev ::= e then(val ev :=:  x | one | zero),
s)    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)         v × u ↑ const(ec ::= v ; value of ec - 1) × udash
         esac,
```

11.11. continued

```
aa)    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 real value of = (form e) :
aj)   begin ref const ec ; ref var ev ; ref triple et ; ref call ef;
ak)   if ec ::= e then value of ec,
al)      ev ::= e then value of ev,
am)      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)      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, b, x;
ay)   start here:
az)   name of a := "a" ; name of b := "b" ; name of x := "x".
ba)   read((value of a, value of b, value of x));
bb)   f := a + x / (b + x) ; g := (f + one) / (f - one);
bc)   print((value of a, value of b, value of x,
            value of(derivative of(g, c with respect to c x)))
bd) end example.
```

{And what impossiblity would slay
in common sense, sense saves another way.
All's well that ends well, William Shakespeare.}