Printed at the Mathematical Centre at Amsterdam, 49, 2nd Boerhaavestraat, The Netherlands.

The Mathematical Centre, founded the 11th of February 1946, is a non - profit institution aiming at the promotion of pure mathematics and its applications, and is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.) and the Central Organization for Applied Scientific Research in the Netherlands (T.N.O.), by the Municipality of Amsterdam and by several industries.

DRAFT REPORT ON THE ALGORITHMIC LANGUAGE

ALGOL 68

A. VAN WIJNGAARDEN (EDITOR),

B. J. MAILLOUX, J. E. L. PECK

AND C. H. A. KOSTER

COMMISSIONED BY
WORKING GROUP 2·1 ON ALGOL
OF THE
INTERNATIONAL FEDERATION
FOR INFORMATION PROCESSING

SUPPLEMENT
TO ALGOL BULLETIN 26

MATHEMATISCH CENTRUM

MR 93

SECOND PRINTING. MARCH 1968

Contents

PP. Provisional Prologue

PP.1. History of the Draft Report

{Habent sua fata libelli.
De litteris, Terentianus Maurus.}

a) Working Group 2.1 on ALGOL of the International Federation for Information Processing has discussed the development of "ALGOL X", a successor to ALGOL 60 [3] since 1963. At its meeting in Princeton in May 1965, WG 2.1 invited written descriptions of the language based on the previous discussions. At the meeting near Grenoble in October 1965, three reports describing more or less complete languages were amongst the contributions, by Niklaus Wirth [5], by Gerhard Seegmüller [4] and by Aad van Wijngaarden [6]. In [4] and [5], the descriptional technique of [3] was used, whereas [6] featured a new technique for language design and definition. Another significant contribution was a paper by Tony Hoare [2].

b) At meetings in Kootwijk in April 1966, Warsaw in October 1966 and Zandvoort near Amsterdam in May 1967, a number of successive approximations to a final report were submitted by a team working in Amsterdam, consisting first of A. van Wijngaarden and Barry Mailloux [7], later reinforced by John Peck [8], and finally by Kees Koster. A rather complete version [9] was used during a course on ALGOL 68 held in Amsterdam in the end of 1967. This course served as a test case and the present Draft Report was made on the basis of it using the experience of explaining the language to a skilled audience.

c) The authors acknowledge with pleasure and thanks the whole-hearted cooperation, support, interest, criticism and violent objections from members of WG 2.1 and many other people interested in ALGOL {Rev. 3.15, 16}. Deserving special mention are Jan Garwick, Jack Merner, Peter Ingerman and Manfred Paul for [1] and above all Miss Hetty Schuuring for still smiling after several years of most demanding typing of a continuously varying manuscript. An occasional choice of a, not inherently meaningful, identifier in the sequel may compensate for not mentioning more names in this section.

d) The dogmatic, perhaps pedantic, approach of the authors, and the many errors they made, caused this Draft Report to appear late; they are convinced, however, that their approach is the right one.

PP.2. Membership of the Working Group

{Verum homines notos sumere odiosum est.
Pro Roscio Amerino, M.T. Cicero.}

At this moment, the members of WG 2.1 are:

F.L. Bauer, H. Bekić, L. Bolliet, E.W. Dijkstra, F.G. Duncan, A.P. Ershov, J.V. Garwick, A. Grau, C.A.R. Hoare, P.Z. Ingerman, E.T. Irons, C. Katz, I.O. Kerner, P.J. Landin, S.S. Lavrov, H. Leroy, J. Loeckx, B.J. Mailloux, A. Mazurkiewicz, J. McCarthy, J.N. Merner, S. Moriguti, P. Naur, M. Nivat, M. Pacelli, M. Paul, J.E.L. Peck, W.L. van der Poel (Chairman), B. Randell D.T. Ross, K. Samelson, G. Seegmüller, W.M. Turski (Secretary), A. van Wijngaarden, N. Wirth, M. Woodger and N. Yoneda.

PP.3. Distribution of the Draft Report

{A perfect judge will read each work of wit
with the same spirit that its author writ.
An Essay on Criticism, A. Pope.}

a) The Draft Report is, on request of WG 2.1, distributed as a supplement to ALGOL Bulletin 26 in order that all people interested in ALGOL have insight in the Draft Report and can send remarks to improve the final Report. These remarks should be sent to:

EDITOR ALGOL 68,
Mathematisch Centrum,
2e Boerhaavestraat 49,
AMSTERDAM-O, HOLLAND.

b) All remarks will be considered, but not necessarily individually answered. If they are received in time and if they seem relevant, then they will be taken into account in drafting the final Report, which will be submitted for approval to WG 2.1, Technical Committee 2 on Programming Languages and the General Assembly of IFIP.

c) It is pointed out that anyone, even an author, is invited to send remarks. In order to make it at all possible to review all incoming remarks, one is, however, requested to use, as far as feasible, the method of indicating errata given in the Ephemeral Epilogue; this method will be clear after reading at least a part of the Draft Report.

PP.3. continued

d) Before reading all of the Draft Report, one should first read Sections EE 2.5, 6.

PP.4. References

[1] J.V. Garwick, J.N. Merner, P.Z. Ingerman and M. Paul, Report of the ALGOL-X-I-O Subcommittee, WG 2.1 Working Paper, July 1966.

[2] C.A.R. Hoare, Record Handling, WG 2.1 Working Paper, October 1965; also AB.21.3.6, November 1965.

[3] P. Naur (Editor), Revised Report on the Algorithmic Language ALGOL 60, Regnecentralen, Copenhagen, 1962, and elsewhere.

[4] G. Seegmüller, A Proposal for a Basis for a Report on a Successor to ALGOL 60, Bavarian Acad. Sci., Munich, October 1965.

[5] N. Wirth, A Proposal for a Report on a Successor of ALGOL 60, Mathematisch Centrum, Amsterdam, MR 75, August 1965.

[6] A. van Wijngaarden, Orthogonal Design and Description of a Formal Language, Mathematisch Centrum, Amsterdam, MR 76, October 1965.

[7] A. van Wijngaarden and B.J. Mailloux, A Draft Proposal for the Algorithmic Language ALGOL X, WG 2.1 Working Paper, October 1966.

[8] A. van Wijngaarden, B.J. Mailloux and J.E.L. Peck, A Draft Proposal for the Algorithmic Language ALGOL 67, Mathematisch Centrum, Amsterdam, MR 88, May 1967.

[9] A. van Wijngaarden, B.J. Mailloux and J.E.L. Peck, A Draft Proposal for the Algorithmic Language ALGOL 68, Mathematisch Centrum, Amsterdam, MR 92, November 1967.

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 {8.9} 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 of the mode of all nonlocal quantities is provided; see the remarks after 2.3.c.

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.

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.

0.2.1. continued

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 or character 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; the ALGOL 68 concept "format" has no ALGOL 60 counterpart.

f) In contrast with plain values and multiple and structured values composed of plain values only, the significance of a name, routine or format or of a multiple or structured value composed of names, routines or formats, possibly amongst other values, is, in general, dependent on the context in which it appears. Therefore, the use of names, routines and formats 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 efficien and powerful parameter mechanism.

2.2. continued

Moreover, in ALGOL 68, a "mode-declaration" permits the construction new modes from already existing ones. In particular, the modes of ltiple values and structured values may be defined this way; in addition, union of modes may be defined for use (e.g. in an identity-declaration lowing each value referred to by a given name to be of one of the ember ~~nstituent~~ modes.

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

2.3. Dynamic storage allocation in ALGOL 68

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

2.4. Collateral elaboration in ALGOL 68

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

.2.5. Standard declarations in ALGOL 68

he ALGOL 60 standard functions are all included in ALGOL 68 along ith many other standard declarations. Amongst these are "environment nquiries", which make it possible to determine certain properties of n implementation, and "transput" declarations, which make it ossible, at run time, to obtain data from and to deliver results to xternal 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.

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~~ excerption 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.

{True wisdom knows
it must comprise
some nonsense
as a compromise,
lest fools should fai
to find it wise.
Grooks, Piet Hein

. Language and metalanguage

.1. The method of description

.1.1. The strict, extended and representation languages

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

) For the definition partly the "English language", and partly a "formal anguage" is used. In both languages, and also in the strict language and he extended language, typographical marks are used which bear no relation o those used in the representation language.

.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" ("abcdefghijklmnopqrstuvwxyz"), possibly interspersed with nonsignificant blanks and/or hyphens.

{Note that those small letters are in a different type font than this sentence. }

b) A "list of notions" either is empty, or is a notion, or consists of a list of notions followed either by a "comma" (",") or 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" (":") and followed by a list of notions, a "direct production" of that notion, and followed by a "point" (".").

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

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

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

1.1.2. continued

{In the production rule
'variable-point numeral : integral part option, fractional part.'
(5.1.2.1.b) of the strict language,
'integral part option, fractional part'
is a direct production of the notion 'variable-point numeral'. A terminal production of this same notion is
'digit zero symbol, point symbol, digit one symbol'.
The notion 'digit 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, 4). }

1.1.3. The syntax of the metalanguage

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

{NOTE THAT THOSE CAPITAL LETTERS ARE IN A DIFFERENT TYPE FONT THAN THIS SENTENCE. }

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 and followed by a list of metanotions, a direct production of that metanotion, and followed by a point.

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

e) A terminal production of a metanotion is a production of that metanotion which is a ~~notion~~, possibly empty, sequence of small letters.

1.1.3. continued

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

1.1.4. The production rules of the metalanguage

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

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

Step 2: A number of production rules for the metanotion 'ALPHA' {1.2.1.n}, each of whose direct productions is another small letter, may be added.

{For instance, the rule
'TAG : LETTER ; TAG LETTER ; TAG DIGIT.',
from 1.2.1.1 is replaced by the rules
'TAG : LETTER.' and 'TAG : TAG LETTER ; TAG DIGIT.',
and the second of these is replaced by
'TAG : TAG LETTER.' and 'TAG : TAG DIGIT.'
thus resulting in three rules from the original one.

The reader may find 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 Step 1 of 1.1.4 ;

1.1.5. continued

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 the metanotions of 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 that metanotion, throughout a copy of the given rule, by that terminal production, whereupon the given rule is discarded and Step 2 is taken; otherwise, the given rule is a production rule of the strict language.

Step 3: A number of production rules for the notions 'other mode indication' and other operator indication {4.2.1.b, e, f} each of whose direct productions is a ~~symbol~~ different basic token {3.0.1.a} from any other symbol may be added.

Step 4: A number of production rules may be added for the notions 'other comment item' {3.0.9.c} and 'other string item' {5.3.1.b} each of whose direct productions is a symbol different from any character-token with the restrictions that no other-comment-item is the comment-symbol and no other-string-item is the quote-symbol.

{The rule
'actual LOWPER bound : strict LOWPER bound.'
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.' and
'actual upper bound : strict upper bound.'.
Note that
'actual lower bound : strict upper bound.'
is not a production rule of the strict language, since the replacement of the metanotion 'LOWPER' by one of its productions must be consistent throughout. Since some metanotions have an infinite number of terminal productions, the number of notions of the strict language is infinite and the number of production rules for a given notion may be infinite; moreover, since some metanotions have terminal productions of infinite length, some notions are infinitely long. For examples see 4.1.1. 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 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 notion ending with 'symbol', ~~in which case~~ (and) 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~~ (and) it then stands for any terminal production of itself {e.g., "number-token" (3.0.3.b) stands for 'digit zero symbol', 'digit one symbol', 'digit two symbol', 'digit three symbol', 'digit four symbol', 'digit five symbol', 'digit six symbol', 'digit seven symbol', 'digit eight symbol', 'digit nine symbol', 'point symbol' or 'times ten to the power symbol'}, or

iii) a notion whose production rule(s) do(es) begin with an asterisk, ~~in which case~~ (and) it then stands for any terminal production of any of its direct productions {e.g. "trimscript" (8.4.1.1) stands for any terminal production of 'trimmer option' or 'subscript'.}, or

v) a paranotion followed by "s", or a paranotion ending with "y" in which "y" has been replaced by "ies", ~~in which case~~ (and) it then stands for some number of the terminal productions stood for by that paranotion {e.g. "trimscripts" stands for some number of terminal productions of 'trimmer option' and/or 'subscript', and "primaries" stands for some number of terminal productions stood for by 'primary'.}, or

) a paranotion whose first small letter has been replaced by the corresponding capital letter, ~~in which case~~ (and) it then stands for the terminal productions stood for by that paranotion before the replacement

1.1.6. continued 1

{e.g. "Identifiers" stands for what 'identifiers' stand for}, or

vi) a paranotion in which a "mode", i.e. a terminal production of 'MODE', has been omitted, ~~in which case~~ (and) 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, when not under Syntax or between apostrophes, are printed with hyphens instead of spaces. 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 1.1.6.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) 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 sequences of symbols by means of sentences in the English language, as a series of processes (the "elaboration" of those sequences of symbols as terminal productions of given notions), each causing a specific effect. Any of these processes may be replaced by any process which causes the same effect.

f) The "preelaboration" of a sequence of symbols as a terminal production of a given notion consists of its elaboration as terminal production of the notion which is a direct production of the given notion and of which it is a terminal production; except as otherwise specified, the elaboration of a sequence of symbols as terminal production of a given notion is its preelaboration as terminal production of that 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). }

1.1.6. continued 2

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

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)* by 9.2.c and 9.2.d.}

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}.

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

1.2. The metaproduction rules

1.2.1. Metaproduction rules of modes

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

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

1.2.2. Metaproduction rules associated with modes

a) PRIMITIVE : integral ; real ; boolean ; character ; format.
b) ROWS : row of ; row of ROWS.
c) ROWSETY : ROWS ; EMPTY.
d) ROWWSETY : ROWSETY.
e) NONROW : TYPE ; reference to MODE ; UNITED.
f) REFETY : reference to ; EMPTY.
g) NONREF : TYPE ; row of MODE ; UNITED.
h) NONPROC : PLAIN ; structured with FIELDS ;
procedure with PARAMETERS DELIVETY ; row of MODE ; UNITED ;
reference to NONPROC.
i) LMODE : MODE.
j) RMODE : MODE.
k) MODETY : MODE ; EMPTY.
l) LMODESETY : MODES and ; EMPTY.
m) RMODESETY : and MODES ; EMPTY.
n) LFIELDSA : FIELDS and a ; a.
o) RFIELDSETY : and FIELDS ; EMPTY.
p) COMPLEX : structured with a real named letter r letter e
and a real named letter i letter m.
q) STRING : row of character.
r) BITS : row of boolean.
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 : ~~FORCED ; hip~~ HIPETY FORCED.
h) FORCED : adapted ; adjusted ; arrayed ; called ; depressed ;
expressed ; fitted ; peeled ; united ; widened.
i) FORCETY : FORCED ; EMPTY.
j) HIPETY : hip ; EMPTY.

1.2.4. Metaproduction rules associated with formulas

a) COERCEND : MODETY FORM.
b) FORM : ADIC formula ; cohesion ; confrontation.
c) ADIC : PRIORITY ; monadic.
d) PRIORITY : priority NUMBER.
e) NUMBER : one ; TWO ; THREE ; FOUR ; FIVE ; SIX ; SEVEN ; EIGHT ;
NINE.
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) OPERATIVE : procedure with a RMODE parameter DELIVETY ;
procedure with a LMODE parameter and a RMODE parameter DELIVETY.

1.2.5. Other metaproduction rules

a) VIRACT : virtual ; actual.
b) VICTAL : VIRACT ; formal.
c) LOWPER : lower ; upper.
d) ALEPH : ALEPH letter f.
e) ANY : sign ; zero ; digit ; point ; exponent ; complex ; character ;
suppressible ANY ; replicatable ANY.
f) NOTION : ALPHA ; NOTION ALPHA.
g) SEPARATORETY : comma symbol ; go on symbol ; completer ; sequencer ;
EMPTY.
h) SELERATOR : selection ; selector ; declarator.

{Rule f 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.b, c, d, e, f. Rule d yields an infinite sequence; for the use of this metanotion, see 5.5.1.6.a. }

{"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 representation language. In these examples, identifiers occur out of context from their defining occurrences. Unless otherwise specified, these occurrences 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 x or y = ref real expr(random < .5 | x | y) ;
proc ncos = (int i) real : cos(2 × pi × i/n) ;
proc nsin = (int i) real : sin(2 × pi × i/n) ;
proc g = (real u) real : (arctan(u) - a + u - 1) ;
proc stop = expr(1:1) ;
exit: princeton: grenoble: kootwijk: warsaw: zandvoort: amsterdam: x :=
```

For serial-declarations see 6.1.1.a, For label-sequence-options see 3.0.1.b,e and 6.1.1.g,

2. The computer and the program

2.1. Syntax

a) program : open, standard declarations, library declarations option, particular program, close.

b) standard declarations : serial declaration, go on symbol.

c) library declarations : serial declaration, go on symbol.

d) particular program : label sequence option, CLOSED statement.

{For opens and closes see 6.4.1.b,c, for standard-declarations see Chapter 10, for collateral-statements see 6.3.1.b, for closed-statements see 6.4 and for conditional-statements see 6.5. 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 explained in terms of a hypothetical "computer" which performs a set of "actions" {2.2.5}, the elaboration of the program {2.3.a}. The computer deals with a set of "objects" {2.2.1} between which, at any given time, certain "relationships" {2.2.2} may "hold".

2.2.1. Objects

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

2.2.2. Relationships

a) Relationships ~~are~~ 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.

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

2.2.2. continued 1

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

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

f) An identifier (operator) may possess a value ({more specifically} a "routine" {2.2.3.4}). This relationship is caused to hold by the elaborati of an "identity-declaration" {7.4} ("operation-declaration" {7.5}) and ceases to hold upon the end of the elaboration of the smallest serial-clause {6.1.1.b} containing that declaration.

g) An external object other than an identifier or operator {e.g. an expression (6.0.1.c)} considered as terminal production of a given notion may be caused to possess a value by its elaboration as terminal productio of that notion, and continues to possess that value until the next elabor if any, of the same occurrence of that external object is "initiated", whereupon it ceases to possess that value.

h) The relationships between internal objects {values} are: "to be of the same mode as", "to be equivalent to", "to be smaller than", "to be a component of" and "to refer to".

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

j) A value may be equivalent to another value {2.2.3.1.d, f} and a value may be smaller than another value {10.2.2.a, 10.2.3.a}. If one of these relationships is defined at all for a given pair of values, then either it does not hold, or it does hold and is permanent.

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

l) Any "name" {2.2.3.5}, except "nil" {2.2.3.5.a}, refers to one instance of another value. This relationship {may be caused to hold by an "assignment" (8.8.2.c) of that value to that name and} continues to hold until another instance of a value is caused to be referred to by that name. The words "refers to an instance of" are often shortened in the sequel to "refers to".

2.2.3. Values

Values are

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

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

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

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

2.2.3.1. Plain values

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

b) An arithmetic value has a "length number", i.e. a positive integer characterising the degree of discrimination with which the value is kept in the computer. The number of integers (real numbers) of given length number that can be distinguished increases with the length number up to a certain length number, ~~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}. For each pair of integers of the same length number, a third integer of that length number may exist, the first integer "minus" the other one {10.2.2.g}. Finally, for each pair of real numbers of the same length number, three real numbers of that length number may exist, the first real number minus ("times", "divided by") the other one {10.2.3.g, l, m} these real numbers are obtained "in the sense of numerical analysis", i.e. by performing the operations known in mathematics by these terms on real numbers which may deviate slightly from the given ones {; this deviation is ~~not defined~~ left undefined in this Report}.

d) Each integer of given length number is equivalent to a real number of that length number. Also, each integer (real number) of given length number is equivalent to an integer (real number) whose length number is greater by one. These equivalences permit the "widening" {8.2.5} of an integer into a real number and the increase of the length number of an integral or real number. The inverse transformations are only

2.2.3.1. continued

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 of length number one; this relationship is defined only in so far that different characters have different integral equivalen

2.2.3.2. Structured values

{Yea, from the table of my memory
I'll wipe away all trivial fond record
Hamlet, William Shakespea

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 specific integer, its "index".

b) The descriptor consists of an "offset", c, and some number, $n \geq 0$, of "quintuples" $(l_i, u_i, d_i, s_i, t_i)$ of integers, $i = 1, \ldots, n$; l_i is ~~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

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

The descriptor "describes" ~~an element iff~~ each element for which there exists an n-tuple $(r_1, \ldots, r_n)$ of integers satisfying $l_i \leq r_i \leq u_i$ for all $i = 1, \ldots, n$ such that the element is selected by

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

{In a given instance of a multiple value, a state which is 0(1) indicate that the given value can (cannot) be "superseded" (8.8.2.a) by an instance of a multiple value in which the bound corresponding to the state differs from that in the given value. }

{ To the name referring to a given multiple value a state of which is 1, no multiple value can be assigned (8.8.2.c Step4) in which the bound corresponding to that state differs from that in the given value.}

2.2.3.3. continued

c) A subvalue of a given multiple value is a multiple value referred to by the value of a slice {8.4} the value of whose constituent whole {8.4.1.a, c} refers to the given multiple value.

2.2.3.4. Routines and formats

A routine (format) is a sequence of symbols which is the same as some closed-clause {6.4.1.a} (format-denotation {5.5}).

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

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

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

2.2.4. Modes and scopes

2.2.4.1. Modes

a) Each instance {2.2.1} of a value is of one specific mode {1.1.6.c.vi} which is a terminal production of 'NONUNITED' {1.2.1.b}; furthermore, all instances of a given value other than nil {2.2.3.5.a} are of one same mode.

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

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

2.2.4.1. continued

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

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

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

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

h) A given mode is "adjusted (united) from" a second mode if the notion consisting of that second mode followed by 'cohesion' is a production of the notion consisting of 'adjusted' ('united') followed by the given mode followed by 'cohesion' {see 8.2}.

{e.g. The mode specified by *real* is adjusted from the mode specified by *ref real*, and that specified by *union(int, real)* is united from those specified by *int* and *real*. }

i) A mode is "related to" a second mode if both modes are adjusted from one same mode {see 4.4.2.a and 4.4.3.c.}.

{e.g. The modes specified by *real*, *ref real*, *union(int, real)* and *proc real* are all related to one another since all are adjusted from 'real' and those specified by *proc[]real* and *[]ref real* are also related to one another.}

j) A given mode is "structured from" a second mode if it begins with 'structured with' and the mode between 'a' and 'named' in one of its portrayals {d} is or is structured from that second mode.

{e.g. In the context of the declarations

struct a = (aa, bb) and

struct b = (aa, ref real r) ,

the mode specified by *a* is structured from those specified by *a* and *b*, whereas the mode specified by *b* is structured from that specified by *a*, and hence also from itself.}

Two modes are "related" to one another if they are the same, or if one of them is adjusted from a mode related to the other one, or if both begin with 'row of' and the modes obtained by deleting the initial 'row of' from each of them are

2.2.4.2. Scopes, inner and outer scopes

a) Each value has one specific scope. Each instance of 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 or format possessed by a given denotation {5.4, 5.5} is the smallest range {4.1.1.e} containing a defining {4.1.2.a} (indication-defining {4.2.2.a}, operator-defining {4.3.2.a}) occurrence of an identifier (indication, operator), if any, applied but not defined (indication-applied but not indication-defined, operator-applied but not operator-defined) within that denotation, and, otherwise, the program, and
that of a name is some {8.5.2.c} range.

c) The inner (outer) scope of an instance of a value possessed by an external object whose value can have only one scope is that scope.

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

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

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

haven't been invented just yet.
Through the Looking Glass, Lewis Carroll.}

) The elaboration of a program is the elaboration of the closed-
tatement {6.4.1.a} consisting of the same sequence of symbols.
In this Report, the Syntax says which sequences of symbols are programs,
nd the Semantics which actions are performed by the computer when
laborating a program. Both Syntax and Semantics are recursive.}

) In ALGOL 68, a specific notation for external objects is used which,
ogether with its recursive definition, makes it possible to handle and
o distinguish between arbitrarily long sequences of symbols, to distinguish
etween arbitrarily many different values of a given mode (except 'boolean')
nd to distinguish between arbitrarily many modes, which allows arbitrarily
ny objects to occur in the computer and which allows the elaboration
a program to involve an arbitrarily large, not necessarily finite,
mber of actions.
is is not meant to imply that the notation of the objects in the
mputer is that used in ALGOL 68 nor that it has the same possibilities.
is, on the contrary, not assumed that the computer can handle
bitrary amounts of presented information. It is not assumed that these
o notations are the same or even that a one-to-one correspondence
ists between them; in fact, the set of different notations of objects
a given category may be finite. It is not assumed ~~that the number of
jects and relationships that can be established is sufficient to cope
h the requirements of a given program nor~~ that the speed of the
puter is sufficient to elaborate a given program within a prescribed
se of time, nor that the number of objects and relationships that can be
ablished is sufficient to elaborate it at all.
A model of the hypothetical computer, using a physical machine, is
d to be an "implementation" of ALGOL 68, if it does not restrict the
of the language in other respects than those mentioned above.
thermore, if a language is defined whose particular-programs are
particular-programs of ALGOL 68 and have the same meaning, then
t language is called a "sublanguage" of ALGOL 68. A model is said
e an implementation of a sublanguage if it does not restrict the
of the sublanguage in other respects than those mentioned above.

{A sequence of symbols which is not a program but can be turned
one by a certain number of deletions or insertions of symbols and
a smaller number could be regarded as a program with that number o
syntactical errors. Any program that can be obtained by performing
number of deletions or insertions may be called a "possibly intend
program. Whether a program or one of the possibly intended program
~~the process its writer in fact intended to describe~~ has the effect its author in fact intended it to have is a matter wh
outside of this Report. }

{In an implementation, the particular-program may be "compiled",
translated into an "object program", ~~written~~ in the code of the phy
machine. Under circumstances, it may be advantageous to compile par
the particular-program independently, e.g. parts which are common t
several particular-programs.
If such a part contains occurrences of identifiers (indications, op
whose defining (indication-defining, operator-defining) occurrences
(Chapter 4) are not contained in that part, then compilation into a
efficient object program may be assured by preceding the part by a
of formal-parameters (5.4.1.e) (mode-declarations (7.2) or priority
declarations (7.3), captions (7.5.1.b)) containing those defining
(indication-defining, operator-defining) occurrences.}

3.0. Syntax

3.0.1. Introduction

a)* basic token : letter token ; denotation token ; action token ;
declaration token ; syntactic token ; sequencing token ;
hip token ; extra token ; quote symbol ; comment symbol ; other mode indication ; other operator indication.

b) NOTION option : NOTION ; EMPTY.

c) chain of NOTIONs separated by SEPARATORETYs : NOTION ;
NOTION, SEPARATORETY, chain of NOTIONs separated by SEPARATORETYs.

d) NOTION list : chain of NOTIONs separated by comma symbols.

e) NOTION sequence : chain of NOTIONs separated by EMPTYs.

f) NOTION pack : open symbol, NOTION, close symbol.

{Examples:

a) *a* ; *0* ; + ; *int* ; *if* ; . ; *nil* ; *for* ; " ; *c* ; primitive ; ?;

b) *0* ; ;

c) *0, 1, 2* ;

d) *0* ; *0, 1, 2* ;

e) *0* ; *000* ;

f) *(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 hip-tokens see 3.0.8, and for extra-tokens see 3.0.9 and for other-mode-indications and other-operator-indications see 1.1.5. Step3.}

3.0.2. Letter tokens

a) letter token : LETTER.

b) LETTER : LETTER symbol.

{Examples:

a) *a* ; π (see 1.1.4.a Step2) }

{Letter-tokens are constituents of identifiers (4.1.1.a), field-selectors (7.1.1.i), format-denotations (5.4.5) and row-of-character-denotations (5.3). }

a) denotation token : number token ; true symbol ; false symbol ;
formatter 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) DIGIT : DIGIT symbol.

e) flipflop : flip symbol ; flop symbol.

{Examples:

a) *1* ; *true* ; *false* ; *f* ; *expr* ; : ; *1* ; , ; . ;

b) *1* ; . ; 10 ;

c) *1* ;

e) *1* ; *0* }

{Denotation-tokens are constituents of denotations (Chapter 5). Some denotation-tokens may, by themselves, be denotations, e.g. the digit-token *1*, whereas others, e.g. the expression-symbol, serve only to construct denotations. }

3.0.4. Action tokens

a) action token :
operator token ; equals symbol ; value of symbol ; confrontation token.

b) operator token : or symbol ; and symbol ; not symbol ;
differs from symbol ; is less than symbol ; is at most symbol ;
is at least symbol ; is greater than symbol ; plusminus ;
times symbol ; over 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 ; minus and becomes symbol ; plus and becomes symbol
times and becomes symbol ; over and becomes symbol ;
modulo and becomes symbol ; prus and becomes symbol ; up symbol ;
down symbol.

c) plusminus : plus symbol ; minus symbol.

d) confrontation token : becomes symbol ; conforms to symbol ;
conforms to and becomes symbol ; is symbol ; is not symbol.

3.0.4. continued

{Examples:

a) + ; = ; *val* ; := ;

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

c) + ; - ;

d) := ; :: ; ::= ; :=: ; :≠: }

{Operator-tokens are constituents of formulas (8.1). An operator-token may be caused to possess an operation by the elaboration of an operation-declaration (7.5).
Confrontation-tokens are constituents of confrontations (8.0.1.c). }

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 ; priority symbol ; operation symbol.

{Examples:

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

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

3.0.6. Syntactic tokens

a) syntactic token : open ; 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 ; end symbol, TAG.~~

{Examples:

a) (;) ; *elem* ; *par* ; [;] ; : ; : ; *if* ; *then* ; *fi* ; *of* ; : }

~~b) (; *begin* ;~~

~~c)) ; *end* ; *end zero*~~}

{Syntactic-tokens separate external objects 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 of elaboration (6.1.2.c). }

3.0.8. Hip tokens

a) hip token : skip symbol ; nil symbol.

{Examples:

a) *skip* ; *nil* }

{Hip-tokens function as skips (6.2.1.e) and nihils (8.3.1.e). }

3.0.9. Extra tokens and comments

a) extra token : for symbol ; from symbol ; by symbol ; to symbol ;
while symbol ; do symbol ; then if symbol ; else if symbol ;
case symbol ; in symbol ; esac symbol ; plus i times symbol.

b) comment : comment symbol, comment item sequence option, comment symbol.

c) comment item : character token ; ~~quote symbol ;~~ other comment item.

d) character token : letter token ; number token ; plus i times symbol ;
open symbol ; close symbol ; space symbol ; comma symbol.

{Examples:

a) *for* ; *from* ; *by* ; *to* ; *while* ; *do* ; *thef* ; *elsf* ; *case* ; *in* ; *esac* ; *i* ;

b) *c with respect to c* ;

c) *w* ; ~~"~~ ? ;

d) *a* ; *1* ; *i* ; (;) ; . ; , }

{For other-comment-items see 1.1.5. Step 4. }

{Extra-tokens and comments may occur in constructions which, by virtue of the extensions of Chapter 9, stand for constructions in which no extra-tokens or comments occur. Thus, a program containing an extra-token or a comment is necessarily a program in the extended language, but ~~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*

symbol	representation
true symbol	*true*
false symbol	*false*
formatter symbol	*f*
expression symbol	*expr*
parameter symbol	: *expr*
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	= *eq*
differs from symbol	≠ *ne*
is less than symbol	< *lt*
is at most symbol	≤ *le*
is at least symbol	≥ *ge*
is greater than symbol	> *gt*
plus symbol	+
minus symbol	-
times symbol	× *
over symbol	/
quotient symbol	÷ *quotient*
modulo symbol	÷: *mod*
absolute value of symbol	*abs*
lengthen symbol	*leng*
shorten symbol	*short*
round symbol	*round*
sign symbol	*sign*
entier symbol	*entier*
odd symbol	*odd*

symbol	representation		
representation symbol	*repr*		
real part of symbol	*re*		
imaginary part of symbol	*im*		
conjugate symbol	*conj*		
binal symbol	*bin*		
to the power symbol	↑	*power*	**:
minus and becomes symbol	*minus*		
plus and becomes symbol	*plus*		
times and becomes symbol	*times*		
over and becomes symbol	*over*		
modulo and becomes symbol	*modb*		
prus and becomes symbol	*prus*		
up symbol	*up*		
down symbol	*down*		
value of symbol	*val*		
becomes symbol	:=		
conforms to symbol	::	*ct*	
conforms to and becomes symbol	::=	*ctb*	
is symbol	:=:	*is*	
is not symbol	:≠:	⊧: *is not*	*isnot*

d) Declaration tokens

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

symbol	representation
complex symbol	*compl*
bits symbol	*bits*
string symbol	*string*
mode symbol	*mode*
priority symbol	*priority*
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	:	

f) Sequencing tokens

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

3.1.1. continued 4

g) Hip 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*	
then if symbol	|:	*thef*
else if symbol	]:	*elsf*
case symbol	(	*case*
in symbol	|	*in*
esac symbol	)	*esac*
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 . }

3.1.2. continued

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

c) Representations of other letter-tokens {1.1.4. Step 2}, other-mode-indications and other-operator-indications {1.1.5. Step 3}, other-comment-items and other-string-items {1.1.5. Step 4} may be added, provided that no ~~two~~ letter-tokens {3.0.2}, or ~~no two~~ indications {4.2}, ~~no two comment-items {3.0.9.c} and no two string-items {5.3.1.b} have the same representation.~~

d) 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. Step 2}.

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

has the same representation as any other basic-token {3.0.1.a}, and that no comment-item {3.0.9.c} (string-item {5.3.1.b}) has the same representation as any other comment-item or the comment-symbol (any other string-item or the quote-symbol).

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 1.2.2.5 and 1.2.1.1 gives rise to an infinity of production rules of the strict language, one for each pair of terminal productions of 'MABEL' and 'TAG'. For example,

'real identifier : letter a letter b.'

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

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

'letter a : letter a symbol.' and

'letter b : letter b symbol.',

yielding

'letter a symbol, letter b symbol'

as a terminal production of '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) A given occurrence of an identifier defines if

i) it follows a formal-declarer {5.4.1.e, ~~7.1.1.a~~},

ii) within some range, it is the textually 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 is contained in a label {6.1.1.g};

otherwise, it "is" applied".

4.1.2. continued

b) If a given occurrence of an identifier is applied, then it may identify a defining occurrence found by the following steps:

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

Step 2: If there exists a smallest range containing the home, then this range, with the exclusion of all ranges contained within it, is called the home and Step 3 is taken {; otherwise, there is no defining occurrenc 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 possess a value (2.2.3) by the elaboration of an identity-declaration (7.4). }

4.2. Indications

4.2.1. Syntax

a)* indication : MODE mode indication ; ADIC indication.

b) MODE mode indication : mode standard ; other mode indication.

c) mode standard : string symbol ;
long symbol sequence option, complex symbol ;
long symbol sequence option, bits symbol.

d)* priority indication : PRIORITY indication.

e) PRIORITY indication : long symbol sequence option, operator token ;
long symbol sequence option, equals symbol ; other operator indication.

f) monadic indication : long symbol sequence option, operator token ;
other operator indication.

g)* adic indication : ADIC indication.

2.1. continued

{Examples:

~~primitive~~ ; *compl* ; *primitive* ;

string ; *long compl* ; *bits* ;

+ ; = ; ? ;

+ ; ~~?~~ ; *long abs* }

mode- and other-operator-indications

{For other indications see 1.1.5. Step 3 and for operator-tokens e 3.0.4.b. }

2.2. Identification of indications

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

If a given occurrence of an indication is indication-applied, then it y identify an indication-defining occurrence of the indication found ing the steps of 4.1.2.b with Step 3 replaced by:

tep 3: If the home contains an indication-defining occurrence of the ndication, then the given occurrence identifies it; otherwise, tep 2 is taken.".

{Indications have no inherent meaning. The indication-defining urrence of an indication establishes that indication as a terminal duction of 'MODE mode indication' (7.2) or 'PRIORITY indication' 3). Monadic-indications have no indication-defining occurrence. }

. Operators

.1. Syntax

operator : procedure with PARAMETERS ADIC operator.

procedure with PARAMETERS DELIVETY ADIC operator :
  procedure with PARAMETERS ADIC operator.

procedure with a LMODE parameter and a RMODE parameter PRIORITY operator :
  PRIORITY indication.

procedure with a RMODE parameter monadic operator : monadic indication.

priority operator : procedure with PARAMETERS PRIORITY operator.

Examples:

;

bs }

4.3.2. Identification of operators

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

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

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

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

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

In the closed-statement

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

the first occurrence of *min* is an indication-defining occurrence of a priority-SIX-indication. The second occurrence *min* is indication-applied and identifies the first occurrence, whereas, at the same textual position, *min* is also operator-defined as a [prrr]-priority-SIX-operator and hence is also a [prr]-priority-SIX-operator (4.3.1.b; i.e. ignoring the mode of t

2. continued

e, if any, which it delivers), where [prr] stands for procedure-
-a-real-parameter-and-a-real-parameter, and [prrr] for [prr]-delivering-
al. The third occurrence of *min* is indication-applied and, as such,
tifies the first occurrence, whereas, at the same textual position,
is also operator-applied, and, as such, identifies the second occurrence
makes it (in view of the identification condition, 4.4.1) a [prr]-
rity-SIX-operator and hence, also because of the identification
ition, a [prrr]-priority-SIX-operator. This identification of the
rity-operator is made because
min occurs in an operation-declaration,
y could be an adjusted-real-priority-SIX-operand,
pi/2 could be an adjusted-real-priority-SEVEN-operand
(since it is a priority-SEVEN-formula),
min is a [prr]-priority-SIX-operator, and
this combination of possibilities satisfies the identification
condition.
this identification of the priority-operator accomplished, we know that
an adjusted-real-priority-SIX-operand and that *pi/2* is an adjusted-
-priority-SEVEN-operand. If the identification condition were not
isfied, then the search for another defining occurrence would be
inued in the same range, or failing that, in a surrounding range. }

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

. Context conditions

proper" program is any program satisfying the context conditions;
meaningful" program is a proper program whose elaboration is defined
this Report. Whether all programs, only proper programs, or only
ningful programs are "ALGOL-68 programs" is a matter for individual
te. {If one chooses only proper programs, then he must consider the
text conditions as syntax which is not written as production rules. }

4.4.1. The identification condition

In a proper program, each applied occurrence of an identifier (each indication-applied occurrence of an indication, each operator-applied occurrence of an operator) which is a terminal production of one or more notions ending with 'identifier' ('indication', operator') is a terminal production of all those same notions at the defining (indication-defining, operator-defining) occurrence, if any, of that identifier (indication, operator). {See the remarks after 4.1.2 and 4.3.2. }

4.4.2. The mode conditions

a) No proper program contains a declarer {7.1} specifying a mode united from {2.2.4.1.h} two modes related {2.2.4.1.i} to one another, or from a mode ~~related to that mode,~~ from which that mode is adjusted.

{e.g., neither the declarer *union(real, ref real)*
nor the mode-declaration {7.2} *mode a = union(real, proc a)*
is contained in any proper program. }

b) No proper program contains a declarer specifying a mode structured from {2.2.4.1.j} itself.

{e.g., no proper program contains the mode-declaration
struct a = (a, real r). }

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

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

4.4.3. The uniqueness conditions

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

b) A given mode-indication {4.2.1.b} is "connected to" a second mode-indication if the actual-declarer following the equals-symbol following t indication-defining occurrence of the given indication ends with an indication which identifies the indication-defining occurrence of the sec indication, or, otherwise is connected to the second indication.

{e.g., in the context of
mode a = ref b ; mode b = proc c ,
the indication *a* is connected to *b* and hence to *c*. }

4.4.3. continued

c) No proper program contains a reach containing two defining occurrences of a given identifier nor two indication-defining occurrences of a given indication.

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

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

(real y ; int y ; sin(3.14)),

(real p ; p : goto p ; sin(3.14)),

(mode a = real ; mode a = bool ; sin(3.14)) and

(mode b = real ; priority b = 6 ; sin(3.14))

is contained in a proper program. }

d) No proper program contains a reach containing two operation-declarations whose first constituent operators are the same indication and all corresponding constituent virtual-~~declarers {7.1.1.b}~~ parameters of whose first constituent tails {7.1.1.w, x, z} ~~specify related {2.2.4.1.i} modes~~ are virtual-declarers specifying modes related to one another {2.2.4.1.i}.

{e.g, neither the closed-expression

(op max = (int a, int b) int : (a > b | a | b) ;

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

nor *(op max = (int a, ref int b) int : (a > b | a | b) ;*

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

is contained in any proper program, but

(op max = (int a, int b) int : (a > b | a | b) ;

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

may be. }

e) No proper program contains a mode-indication which is connected to {4.4.3.b} itself.

{e.g., neither of the mode-declarations

mode a = a and

mode b = ref b nor the pair of declarations

mode c = ref d ; mode d = proc c

is contained in any proper program.}

f) No proper program contains an applied occurrence of an identifier (indication-applied occurrence of a mode-indication or priority-indication, operator-applied occurrence of an operator) which does not identify a defining (an indication-defining, an operator-defining) occurrence.

5. Denotations

5.0.1. Syntax

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

{Examples:

a) *3.14* ; *101* ; *"algol_report"* ; *(bool a, b) bool : (a | b | false)* ; *f5df*

{For plain-denotations see 5.1, for row-of-boolean-denotations see 5.2, for row-of-character-denotations see 5.3, for routine-denotations see 5.4 and for format-denotations see 5.5. }

5.0.2. Semantics

a) A denotation possesses a value; a given denotation always possesses the same value; its elaboration involves no action.

b) The mode of the value possessed by a given denotation is obtained by deleting 'denotation' from that direct production of the notion 'denotation' of which the given denotation is a terminal production. {e.g. The value of *"algol_report"*, which is a production of 'row of character denotation', is of the mode 'row of character'. }

5.1. Plain denotations

5.1.0.1. Syntax

a)* plain denotation : PLAIN denotation.

b) long INTREAL denotation : long symbol, INTREAL denotation.

{Examples:

a) *4096* ; *3.14* ; *true* ;

b) *long 4096* ; *long long 3.141592653589793* }

{For integral-denotations see 5.1.1, for real-denotations see 5.1.2 and for boolean-denotations see 5.1.3. }

5.1.0.2. Semantics

a) A plain-denotation possesses a plain value {2.2.3.1}, but plain values possessed by different plain-denotations are not necessarily different.

5.1.0.2. continued

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

5.1.1. Integral denotations

5.1.1.1. Syntax

a) integral denotation : digit zero ; natural numeral.
b) natural numeral : digit FIGURE, digit token sequence option.

{Examples:
a) *0* ; *4096* ;
b) *1* ; *2* ; *3* ; *123* ; (Note that *00123* and *-7* 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 that integral-denotation in the representation language {1.1.8}.
{See also 5.1.0.2.b}

5.1.2. Real denotations

5.1.2.1. Syntax

a) real denotation : 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 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* ; 1_{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 as many times as there are digit-token in the fractional-part.

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

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

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

5.1.3. Boolean denotations

5.1.3.1. Syntax

a) boolean denotation : true symbol ; false symbol.

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

5.1.3.2. Semantics

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

5.2. Row of boolean denotations

5.2.1. Syntax

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

2.1. continued

{Examples:

101 ; *long 101* }

{For flipflops see 3.0.3.e. }

2.2. Semantics

Let m stand for the number of flipflops in the denotation and n for the lue of *L bits width* {10.1.g}, *L* standing for as many times *long* as there e long-symbols in the denotation.

If m ≤ n, then the value of the row-of-boolean-denotation is a multiple lue {2.2.3.3} whose descriptor has an offset 1 and one quintuple , n, 1, 1, 1) and whose element with index i is a new instance of true alse) if the i-th constituent flipflop is a flip-symbol (a flop-symbol) r i = 1, ..., m and of false for i = m + 1, ..., n {; otherwise, the lue is undefined}.

{If the value of *bits width* is, say, 6, then *101* possesses the same lue as the collateral-expression (true, false, true, false, true, false), t *101* is not a collateral-expression. }

3. Row of character denotations

3.1. Syntax

) STRING denotation :
quote symbol, string item sequence option, quote symbol.
) string item : character token ; quote image ; other string item.
) quote image : quote symbol, quote symbol.

{Note that, since the Syntax nowhere allows row-of-character-denotations o occur following one another, the quote-image can cause no ambiguities. }

{Examples:

) "" ; "a" ; "abcde" ; "a.+.b.""is.a.formula""" ;
) *a* ; "" ; *?* ;
) "" }

{For character-tokens see 3.0.9.d and for other-string-items see .1.5. Step 4. }

5.3.2. Semantics

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

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

5.4. Routine denotations

5.4.1. Syntax

a)* routine denotation : PROCEDURE denotation.

b) procedure with PARAMETERS delivering a MODE denotation :
formal PARAMETERS pack, virtual MODE declarer, parameter symbol, hip adapted MODE primary.

c) procedure with PARAMETERS denotation :
formal PARAMETERS pack, parameter symbol, primary 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, hip adapted MODE primary.

g) procedure denotation : expression symbol, primary 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)* ;
g) *expr(n = 1966 | warsaw | zandvoort)* }

{For hip-adapted-primaries see 8.3.1.a and for primary-statements see 6.2.1.c. }

5.4.2. Semantics

A routine-denotation possesses 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.

{For the use of routines, see 8.1 (formulas), 8.2.1 (unaccompanied-calls) and 8.7 (accompanied-calls). }

5.5. Format denotations

5.5.1. Syntax

a) format denotation :
formatter symbol, format primary list, formatter symbol.
b) format primary : format item ; insertion option, replicator, format primary list pack, insertion option.
c) format item : MODE pattern, insertion option.
d) insertion : literal option, insert sequence ; literal.
e) insert : replicator, alignment, literal option.
f) replicator : replication option.
g) replication : dynamic replication ; integral denotation.
h) dynamic replication : letter n, fitted serial integral expression pack.
i) alignment : letter k ; letter x ; letter y ; letter l ; letter p.
j) literal : STRING denotation option , replicated literal sequence ; STRING denotation.
k) replicated literal : replication, STRING denotation.

{Examples:
a) *fp"table_of"x10a,n(lim-1)(l6x3zd,3x10(2x+.12de+2d"+j×"si+.10de+2d))pf* ;
b) *p"table_of"x10a ; 3x10(2x+.12de+2d"+j×"si+.10de+2d)* ;
c) *l20kc("mon","tues","wednes","thurs","fri","satur","sun")"day"* ;
d) *p"table_of"x ; "day"* ;
e) *p"table_of"* ;
g) *n(lim-1)* ; *10* ;

5.5.1. continued

h) *n(lim-1)* ;
j) *"+j×"* ;
k) *20"_"* }

l) sign mould : loose replicatable zero frame option, sign frame.
m) loose ANY frame : insertion option, ANY frame.
n) replicatable ANY frame : replicator, ANY frame.
o) zero frame : letter z.
p) sign frame : plusminus.
q) suppressible ANY frame : letter s option, ANY frame.
r)* frame : ANY frame.

{Examples:
l) *"="12z+* ;
m) *"="12z* ;
n) *12z* ;
q) *si* ; *10a* }

{Formats (see 5.5.2.a) are used by the formatted transput routines (10.5.4,5) to control "transput", i.e. "input" from and "output" to a "file" (10.5.1).
A format-item is used on output to control the "conversion" of a value to a "string", i.e. a value of mode 'row of character', and, on input, that of a string to a value.
The mode specified by a format-item is that obtained by deleting 'pattern' from that notion ending with 'pattern' whose terminal production is the constituent pattern of that format-item.

Formats have a complementary meaning on input and output; that is, under control of one given format-item:

i) it is possible to convert a given value to a string by means of a formatted output routine, provided the mode specified by the format-item is "output-compatible" with the mode of the given value, and the number of characters specified by the format-item is sufficient (10.5.4) ;
ii) it is possible to convert a given string to a value of a given mode, provided the mode specified by the format-item is "input-compatible" with the mode of the value, the number of elements of the string is

the same as that specified by the format-item, and the individual characters of the string "agree" with the frames of the format-item specifying them (10.5.5) ;

iii) if it is possible to convert a given value to a string and the format-item does not contain a letter-k or letter-y as alignment, and the format-item does not contain any digit-frames or character-frames preceded by letter-s, then it is possible to convert the resulting string (under control of the same format-item) into a value; the resulting value is equal (approximately equal) to the given value if the given value is a string, integer or truth value (is a real value) ;

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

{The value of the empty replicator is one; the value of a replication that is an integral-denotation is the value of that denotation; the value of a dynamic-replication is the value of its constituent fitted-serial-integral-expression if that value is positive, and zero otherwise.
The number of characters specified by a format-item is the sum of the numbers of characters specified by its constituent frames and the number specified by a frame is equal to the value of its preceding replicator.

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

A format-primary which is not a format-item can control the transput of a number of values; this number is at most the value of the constituent replicator times the sum of the numbers of values of which the transput can be controlled by the constituent format-primaries of its constituent format-primary-list-pack.

An insertion is "performed" by performing its constituent literals and/or alignments one after the other.

On output, a (replicated-) literal is "written" (10.5.4.k) on the file, starting from the current position on the line, as many times as the value of the replicator.
On input, a (replicated-) literal is "required" (10.5.5.b) on the file, starting from the current position on the line, as many times as the value of the replicator. If the string possessed by the literal is present then it is skipped; otherwise, the further elaboration is undefined.
An alignment may change the current page count, line count and position on the line of the file as follows: (let n stand for the value of the preceding replicator)

a) letter-k causes the position on the line to be set to n;
b) letter-x causes the position to be incremented by n (10.5.1.2.m);
c) letter-y causes the position to be decremented by n (10.5.1.2.n);
d) letter-l causes the line count to be incremented by n and the position on the line to be reset to one (10.5.1.2.o);
e) letter-p causes the page count to be incremented by n and both line count and position on the line to be reset to one (10.5.1.2.p).

A format-item can be used to "edit" a value as follows:

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

ii) If the format-item contains a sign-mould, then a character specified by the sign-frame will be used to indicate the sign, viz., if the sign-mould contains a minus-symbol and the value is positive (negative), then a space (minus), and, otherwise, a plus (minus). This character is shifted in that part of the string specified by the sign-mould as far to the right as possible across any leading zeroes and those zeroes are replaced by spaces;
e.g., under the sign-mould *4z+*, the string possessed by *"+0003"* is edited into that possessed by *"...+3"*. If the format-item does not contain a sign-mould and the value is negative, then the result is undefined.

iii) Leading zeroes in those parts of the string specified by any remaining zero-frames are replaced by spaces; e.g., under the

5.5.1. continued 4

format-item *zdzd2d*, the integer possessed by *180768* is edited into the string possessed by *"18_168"*.

iv) Suppressed characters are deleted.

A format-item can be used to "indit" a string into a value of a given mode as follows:

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

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

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

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

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

5.5.1.1. Integral patterns

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

b) radix mould : radix, letter r.

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

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

e) digit frame : zero frame ; letter d.

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

5.5.1.1. continued

{Examples:

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

b) *2r ;*

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

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

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

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

i) on output, the value to be output is edited into a string and "transcribed onto" the file by, for all frames occurring in the pattern, first performing the preceding insertion, if any, and then outputting to the file (10.5.1.2.k) that part of the string specified by the frame, and, finally, by performing the insertion, if any, following the last of those frames;

ii) on input, a string is "transcribed from" the file, which string is obtained by, for all frames occurring in the pattern, first performing the preceding insertion, if any, and then, for a frame that is not suppressed, inputting (10.5.1.2.j) from the file as many characters as are specified by the frame; that string is indited into a value; and

~~iii) finally,~~ the insertion, if any, following the last ~~constituent~~ of those frames is performed.

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

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

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

iii) finally, the insertion, if any, following the constituent literal-list-pack is performed. }

5.5.1.2. Real patterns

a) real pattern : sign mould option, real mould ; floating point mould.
b) real mould : integral mould, loose suppressible point frame,
integral mould option ;
loose suppressible point frame, integral mould.
c) point frame : point symbol.
d) floating point mould :
stagnant mould, loose suppressible exponent frame,
sign mould option, integral mould.
e) stagnant mould : sign mould option, INTREAL mould.
f) exponent frame : letter e.

{Examples:
a) *+12d* ; *+d.11de+2d* ;
b) *d.11d* ; *.12d* ;
d) *+d.11de+2d* ;
e) *+d.11d* }

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

5.5.1.3. Boolean patterns

a) boolean pattern :
insertion option, letter b, boolean choice mould option.
b) boolean choice mould :
open symbol, literal, comma symbol, literal, close symbol.

{Examples:
a) *l"result"l4xb* ; *b("","error")* ;
b) *("","error")* }

{If the boolean-pattern does not contain a choice-mould, then the effect of using the pattern is the same as if the letter-b were followed by *("1","0")*.
The insertion, if any, preceding the letter-b is performed, and,
i) on output, if the truth value to be output is true, then the first constituent literal of the constituent choice-mould is performed, and, otherwise, the second;

5.5.1.3. continued

ii) on input, one of the constituent literals of the constituent choice-mould is required on the file; if the first literal is present, then the value true is found; otherwise, if the second literal is present, then the value false is found; otherwise, the further elaboration is undefined;
iii) finally, the insertion, if any, following the constituent choice-mould is performed. }

5.5.1.4. Complex patterns

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

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

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

5.5.1.5. String patterns

a) STRING pattern :
loose replicatable suppressible character frame sequence.
b) character frame : letter a.

{Example:
a) *p"table_of"x10a* }

{On output, a given string is edited into a string and transcribed onto the file;
on input, a string is transcribed from the file and indited into a string.
If the value to be transput is a character, then a string having one element is transput.
A string to be output must have as many elements as the number of characters specified by the format-item. }

5.5.1.6. Transformats

a) structured with a STRING named ALEPH transformat :
hip adapted unitary format expression.

{Example: *(x≥0|f5df|f5d"-"f)* }

5.5.1.6. continued

{For unitary-expressions see Chapter 8. }

{Transformats are used exclusively as actual-parameters of *out* (10.5.4.a) and *in* (10.5.5.a); for reasons of efficiency, the programmer has deliberately been made unable to use them elsewhere by the choice of 'ALEPH' (1.2.5.d).

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

5.5.2. Semantics

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

b) A given transformat is elaborated in the following steps:
Step 1: It is preelaborated {1.1.6.f} ;
Step 2: It is replaced by the format obtained in Step 1, and the thereby resulting format-denotation is considered ;
Step 3: All constituent dynamic-replications {5.5.1.h} of the considered format-denotation are elaborated collaterally {6.3.2.a}, where the elaboration of a dynamic-replication is that of its constituent serial-expression;
Step 4: Each of those dynamic-replications is replaced by that integral-denotation {5.1.1} which possesses the same value as that dynamic-replication if that value is positive, and, otherwise, by a digit-zero ;
Step 5: That row-of-character-denotation {5.3} is considered which would be obtained by replacing, in the considered format-denotation as modified in Step 4, each constituent quote-symbol by a quote-image {5.3.1.c} and ~~each~~ then the textually first and textually last constituent formatter-symbol by a quote-symbol ;
Step 6: A new instance of the value of the considered row-of-character-denotation is made to be the {only} field of a new instance of a structured value {2.2.3.2} whose mode is that obtained by deleting 'transformat' from that notion ending with 'transformat' of which the given transformat is a terminal production ;
Step 7: The considered format-denotation is replaced by the given transformat, and that transformat is made to possess the structured value obtained in Step 6.

6. Phrases

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~~ as 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~~ left undefined in this Report.}

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

5.0.2. continued

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

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

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

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

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

{Clauses are protected in order to allow unhampered definitions of dentifiers, indications and operators within ranges and to permit a eaningful 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 : serial declaration, 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) *l : true ; l1 : l2 : x := a + 1 ; (x > 0 | l3 | x := 1 - x) ; false.*
l3 : y := y + 1 ; true ;

c) *real x ; int i ; ;*

d) *l1 : l2 : x := a + 1 ; (x > 0 | l3 | x := 1 - x) ; false ;*

e) *x := a + 1 ; (x > 0 | l3 | x := 1 - x) ; ;*

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.

6.1.2. continued

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 constituent unitary-phrase;

ii) terminated upon the termination of the elaboration of a constituent unitary-phrase appointing a successor outside the serial-phrase, and that successor {6.2.2.b} 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-~~constituent~~ 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 possessed 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 possessed by *r* is the serial-expression itself. }

6.2. Unitary statements

6.2.1. Syntax

a) unitary statement : formary statement ; MODE confrontation.
b) formary statement : ADIC formula ; called ADIC formula ; NONPROC ADIC formula ; called NONPROC ADIC formula ; primary statement.
c) primary statement : CLOSED statement ; cohesive statement ; called cohesion ; called NONPROC cohesion.
d) cohesive statement : jump ; skip ; statement call ; NONPROC cohesion.
e) skip : skip symbol.
f) jump : go to symbol option, label identifier.

6.2.1. continued

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

{For unitary-declarations see Chapter 7, and for unitary-expressions see Chapter 8.
For confrontations see 8.0.1.d, for formulas see 8.1, for called-formula and 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 statement-calls see 8.7.1.c and for cohesions see 8.3.1.b.

6.2.2. Semantics

a) The elaboration of a skip involves no action.

{For the use of skips as statements and expressions see the remarks after 8.3.2.d. }

b) 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). }

6.3. 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 lis close symbol.

.1. continued

{Examples:

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

(x := 0, y := 1) ; *(x := 0, y := 1, z := 2)* ;

(x, n) ; *(1, 2.3, 4.5)* }

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

.2. Semantics

If a number of constituents of a given terminal production of a notion
"elaborated collaterally", then this elaboration is the collateral
ion {2.2.5} consisting of the {merged} elaborations of these constituent
is:

initiated by initiating the elaboration of each of these constituents ;

) interrupted upon the interruption of the elaboration of any of these constituents ;

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

) terminated upon the termination of the elaboration of any of these constituents, and if that constituent appoints a successor, then this is the successor of the given terminal production.

A collaterał-phrase is elaborated in the following steps, where "m"
nds for the number of its constituent unitary-phrases:

ep 1: Its constituent unitary-phrases are elaborated collaterally {a};
if it is an expression, then Step 2 is taken; otherwise, its elaboration
is complete ;

ep 2: If the values of the constituent unitary-expressions of the
collateral-expression are names {2.2.3.5} one or more of which refers
o an element or subvalue of a multiple value having one or more states
{2.2.3.3.b} equal to zero, or if the values of those unitary-expressions
are multiple values {2.2.3.3} not all of whose corresponding upper (lower)
ounds are equal, then the further elaboration is undefined; otherwise,
Step 3 is taken ;

6.3.2. continued

Step 3: The value of the collateral-expression is a new instance of a multiple value whose mode is that obtained by deleting 'collateral', 'expression' and the terminal production of 'COERCETY' from {the notion which is} that direct production of 'collateral expression' of which the given collateral-expression is a terminal production; this new multiple value is established as follows:

if

the values obtained in Step 1 are not multiple values

then

its element with index "i" is a new instance of the value of the i-th constituent unitary-expression and its descriptor consists of an offset 1 and one quintuple (1, m, 1, 1, 1);

otherwise, those values are multiple values and the elements with indices $(i-1) \times r + j$, $j = 1, \ldots, r$ of the new value, where r stands for the number of elements in one of those values, are the elements of the value of the i-th constituent unitary-expression and the descripto of the new value is a copy of the descriptor of the value of one of the constituent unitary-expressions in which an additional quintuple (1, m, r, 1, 1) has been inserted in front of the old first quintuple, the offset has been set to 1, d_n has been set to 1, and, for $i = n, n-1, \ldots, 2$, the stride d_{i-1} has been set to $(u_i - l_i + 1) \times$

{~~The presence of a~~ A parallel-symbol makes it possible to control the progress of the elaborations of the constituent unitary-phrases by means of the synchronization operations 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 and for opens and closes see 3.0.6.b, c. }~~

b) (; begin

c)) ; end ; end increment}

{ For serial phrases see 6.1.}

e) open: open symbol ; begin symbol.

.4.2. Semantics

The elaboration of a closed-phrase is that of its constituent serial-
rase.

The value of a closed-expression is that, if any, of its constituent
rial-expression.

The elaboration of a closed-phrase which begins with an elementary-
mbol is an elementary action {2.2.5}.

5. Conditional clauses

5.1. Syntax

COERCETY conditional CLAUSE :
  if symbol, COERCETY choice CLAUSE, fi symbol.
COERCETY choice CLAUSE :
  condition, COERCETY then CLAUSE, COERCETY else CLAUSE option.
condition : fitted serial boolean expression.
COERCETY THELSE CLAUSE : THELSE symbol, COERCETY serial CLAUSE.

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

{For serial-clauses see 6.1.1.b.}

5.2. Semantics

A conditional-clause is elaborated in the following steps:
ep 1: Its constituent condition is elaborated ;
ep 2: If the value of that condition is true, then its constituent then-
clause and otherwise its constituent else-clause, if any, is considered ;
ep 3: The clause following the then-symbol or else-symbol of the
considered clause, if any, is elaborated ;
ep 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.

6.5.2. continued

b) The elaboration of a conditional-clause is
i) interrupted (halted, resumed) upon the interruption (halting,
  resumption) of the elaboration of the condition or the considered cla
ii) completed upon the completion of the elaboration of the considered
  clause, if any; otherwise, completed upon the completion of the
  elaboration of the condition; and
iii) terminated upon the termination of the elaboration of the conditio
  or considered clause, and, if one of these appoints a successor, ther
  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* ;
priority plus = *1* ;
int m = *4096* ; *real x* ; *bool complete*(*false*) ;
proc sgn = (*real x*) *int* : (*x* = *0* | *1* | *sign x*) ;
op + = (*real a, b*) *int* : (*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.b} whose successor is a unitary-clause outside that declaration but within the smallest range containing it, then the further elaboration is undefined.

b) An external object {2.2.1} which was caused to possess a value by the elaboration of a declaration ceases to possess that value 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) VIRACT structured with FIELDS declarator :
structure symbol, VIRACT FIELDS declarator pack.
f) VIRACT FIELDS and a FIELD declarator :
VIRACT FIELDS declarator, comma symbol, VIRACT FIELD declarator.
g) VIRACT MODE named TAG declarator :
VIRACT MODE declarer, MODE named TAG selector.
h) MODE named TAG selector : TAG.
i)* field SELERATOR : FIELD SELERATOR ; VIRACT FIELD SELERATOR.

{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, virtual NONROW declarer.
o) VICTAL row of ROWS rower : VICTAL row of rower ~~option~~, comma symbol, VICTAL ROWS rower.
p) VICTAL row of rower : VICTAL lower bound, up to symbol, VICTAL upper bound; EMPTY
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 : hip fitted integral formary.
u) flexible LOWPER bound : integral symbol, integral identifier.

{Examples:
n) [*1 : m, 1 : n*] *real* ;
o) *1 : m, 1 : n* ; , *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, PROCEDURE tail.
w) procedure tail : EMPTY.
x) procedure with PARAMETERS tail : virtual PARAMETERS pack.
y) virtual MODE parameter : virtual MODE declarer.
z) procedure PARAMETY delivering a MODE tail : procedure PARAMETY tail, virtual MODE declarer.

7.1.1. continued 2

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

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

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

{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,
mode cell = *struct*(*string name, ref cell next*) and
mode link = *struct*(*ref link next, ref cell value*)
may both be present in some range.

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 {1.1.2.c} of the notion 'declarer' of which the given declarer is a production.

b) A given declarer is "developed" as follows:

Step: If it is, or contains, a mode-indication which is ~~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} which contains its indication-defining occurrence {4.2.2.b}, and the Step is taken again; otherwise, the development of the declarer ~~is complete~~ has been accomplished.

{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 4 is taken; otherwise, if it now begins with a sub-symbol, then Step 5 is taken; otherwise, if it now begins with a union-of-symbol, then Step 3 is taken; otherwise, a new instance of a value of the mode specified {a} by the given actual-declarer is considered, and Step 8 is taken ;

Step 3: Some mode is considered which does not begin with 'union of' and from which the mode specified by the given actual-declarer is united {2.2.4.1.h}, a new instance of a value of the considered mode is considered, and Step 8 is taken ;

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

7.1.2. continued

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

Step 6: A descriptor {2.2.3.3} is established consisting of an offset 1 and as many quintuples as there are constituent actual-row-of-rowers in the given declarer; if the i-th of these actual-row-of-rowers contains a constituent 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} ; then d_n is set to 1, and, for $i = n, n-1, \ldots, 2$, the stride d_{i-1} is set to $(u_i - l_i + 1) \times d_i$;

Step 7: The descriptor is made to be the descriptor of a multiple value of the mode specified by the given actual-declarer; each of its elements is a new instance of some value of some mode {not beginning with 'union of' and} such that the mode specified by the last constituent virtual-declarer is or is united from {2.2.4.1.h} it; this multiple value is considered ;

Step 8: A name {2.2.3.5} different from all other names and whose mode is 'reference to' followed by the mode specified by the actual-declarer, is created and made to refer to the considered value; this name is 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 9.2b) }

7.2.2. Semantics

The elaboration of a mode-declaration involves no action.

{See 4.4.2. concerning certain mode-declarations which are not contained in proper programs. }

7.3. Priority declarations

7.3.1. Syntax

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

{Example:

a) *priority* + = *6* }

7.3.2. Semantics

The elaboration of a priority-declaration involves no action.

{For a summary of the standard priority-declarations, see the remarks in 8.1.2. }

7.4. Identity declarations

7.4.1. Syntax

a) identity declaration :
formal MODE parameter, equals symbol, actual MODE parameter.
b) actual MODE parameter : MODE transformat ;
hip 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.)

7.4.1. continued

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

{For formal-parameters see 5.4.1.e, for hip-adapted-unitary-expressions see 8.0.1.a, for local-generators see 8.5.1.b and for transformats see 5.5.1

7.4.2. Semantics

An identity-declaration is elaborated in the following steps:
Step 1: Its textually first constituent formal-declarer {in the formal-parameter} is developed {7.1.2.b} ;
Step 2: Its textually last constituent actual-parameter, and all strict-lower-bounds and strict-upper-bounds contained in that formal-declarer, as possibly modified by Step 1, but not contained in any constituent strict-lower-bound or strict-upper-bound of that formal-declarer, are elaborated collaterally {6.3.2.a} ;
Step 3: If the value of that 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 ;
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 that formal-declarer is made to possess a new instance of the value of the corresponding bound in the {multiple} value which is or is referred to by the value of that 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 that formal-declarer which is or is referred to by the value is not the same as that of the corresponding bound in the value of that actual-parameter, then the further elaboration is undefined; otherwise, the identifier following that formal-declarer is made to possess a new instance of the value of that actual-parameter.

All applied occurrences {4.1.2.b} of identifiers in constituent flexible-lower-bounds and flexible-upper-bounds of that formal-declarer are elaborated {8.3.2.a} collaterally; if the value of any of

.4.2. continued

{According to Step 5, the elaboration of the declaration

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

s 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)). }

5. Operation declarations

5.1. Syntax

operation declaration :

OPERATIVE caption, equals symbol, actual OPERATIVE parameter.

OPERATIVE caption :

operation symbol, OPERATIVE tail, OPERATIVE ADIC operator.

{Examples:

op abs = (*real a*) *real* : (a < 0 | -a | a) (see 9.2.f) ;

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

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

{For actual-parameters see 7.4.1.b, for tails see 7.1.1.w, x, z and
operators see 4.3. }

5.2. Semantics

operation-declaration is elaborated in the following steps:

p 1: Its constituent ~~expression~~ actual-parameter is elaborated ;

p 2: The operator preceding its textually first constituent equals-symbol is made
o possess the {routine which is the} value obtained in Step 1.

{The formula (8.1) p ∧ q, where ∧ identifies the operator-defining
urrence of ∧ in the operation-declaration

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

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

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

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

8. Unitary expressions

8.0.1. Syntax

a) COERCETY unitary MODE expression :
COERCETY MODE formary ; COERCETY MODE confrontation.

b) COERCETY MODE formary :
COERCETY MODE ADIC formula ; COERCETY MODE primary.

c) hip FORCED MODE ADIC formula : FORCED MODE ADIC formula.

d) hip FORCED MODE confrontation : FORCED MODE ~~ADIC formula~~ confrontat

e) MODE confrontation : MODE assignation ;
MODE conformity relation ; MODE identity relation.

{Examples:

a) *k* + 1 ; *x* := 3.14 ;

b) *k* + 1 ; *x* ; *nil* ;

c) *i* + *j* (in *x* := *i* + *j*) ;

e) *x* := 3.14 ; *ec* :: *e* (see 11.11.q) ; *val xx* :=: *x or y* }

{For formulas see 8.1, for primaries see 8.3, for assignations see 8.
for conformity-relations see 8.9 and for identity-relations see 8.10. }

8.1. Formulas

8.1.1. Syntax

a)* COERCETY formula : COERCETY ADIC formula DELIVETY.

b) MODE ADIC formula : ADIC formula delivering a MODE.

c) PRIORITY formula DELIVETY : LMODE PRIORITY operand, procedure with a
LMODE parameter and a RMODE parameter DELIVETY PRIORITY operator,
RMODE PRIORITY plus one operand.

d)* operand : MODE ADIC operand.

e) MODE PRIORITY operand :
adjusted MODE PRIORITY formula ; MODE PRIORITY plus one operand.

f) MODE priority NINE plus one operand : MODE monadic operand.

g) MODE monadic operand :
adjusted MODE monadic formula ; hip adjusted MODE primary.

h) monadic formula DELIVETY : dep DELIVETY ; procedure with a RMODE
parameter DELIVETY monadic operator, RMODE monadic operand.

i) dep delivering a MODE :
value of symbol, peeled reference to MODE monadic formula ;
value of symbol, hip peeled reference to MODE primary.

j)* depression : dep DELIVETY.

1.1. continued

{Examples:
) $x + y$; $a + b \uparrow -2$; (priority 6, 6)
) $b \times (a > 0 \mid a \mid$ goto exit) ; $b \uparrow -2$; (priority 7,8)
) 2 ; -2;
) (a > 0 | a | goto exit) ; 2 ;
) val xx ; -2 ;
) val xx }

{For adjusted-formulas see 8.2.0.1.d, for hip-adjusted-primaries and eled-primaries see 8.3.1.a and for peeled-formulas see 8.2.1.1.e. }

1.2. Semantics

A formula other than a depression is elaborated in the following steps:
ep 1: The formula is replaced by a copy of the routine possessed by the constituent operator at its operator-defining occurrence {7.5.2, 4.3.2.b} ;
ep 2: The copy {which is now a closed-expression} is protected {6.0.2.d} ;
ep 3: The skip-symbol {5.4.2.ii} following the equals-symbol following the textually first constituent formal-parameter of the copy is replaced by a copy of the textually first constituent operand of the formula, and if the constituent operator is not a monadic-operator then the skip-symbol following the equals-symbol following the textually second constituent formal-parameter of the copy is replaced by a copy of the textually second constituent operand of the formula ;
ep 4: The elaboration of the copy is initiated; its value, if any, is then that of the formula; if this elaboration is completed or terminated then the copy is replaced by the formula before the elaboration of a successor is initiated.

A depression is elaborated in the following steps:
ep 1: Its constituent peeled-formula or hip-peeled-primary is elaborated ;
ep 2: The value of the depression is a new instance of the value referred to by the name obtained in Step 1.

8.1.2. continued

{The following table summarizes the operator-tokens as declared in the standard-declarations (10.2.0).

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

Observe that the value of *(- 1 ↑ 2 + 4 = 5)* and that of *(4 - 1 ↑ 2 = 3)* both are true, since the first minus-symbol is a monadic-operator whereas the second is dyadic.
Although the Syntax 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.* }

8.2. Coercends

8.2.0.1. Syntax

a)* coercend : FORCETY COERCEND.
b)* FORCED coercend : FORCED COERCEND.
c) adapted COERCEND :
adjusted COERCEND ; widened COERCEND ; arrayed COERCEND.
d) adjusted COERCEND :
fitted COERCEND ; expressed COERCEND ; united COERCEND.
e) fitted COERCEND : COERCEND ; called COERCEND ; depressed COERCEND.

.2.0.1. continued

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

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

{The coercion process may be illustrated by considering the analysis of *random* in *random + x*. According to 10.2.3.i, 10.2.0.a and 8.1.1.b,c, *random + x* is a real-priority-SIX-formula and *random* must therefore be a real-priority-SIX-operand, which may be produced as a procedure-delivering-a-real-identifier (see 10.3.k and 7.4.1.a.) as follows:
real-priority-SIX-operand,
real-priority-SEVEN-operand (8.1.1.e),
real-priority-EIGHT-operand (8.1.1.e),
real-priority-NINE-operand (8.1.1.e),
real-priority-NINE-plus-one-operand (8.1.1.e),
real-monadic-operand (8.1.1.f),
hip-adjusted-real-primary (8.1.1.g),
hip-adjusted-real-cohesion (8.3.1.a),
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.b).
A coercion is derived from the context and is passed on by the Syntax until it meets a coercend (i.e., formula, cohesion or confrontation), where it is activated (i.e., stripped, called, expressed, depressed, united, widened or arrayed). In the above example, the coercion was activated by a called-cohesion which resulted 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 COERCEND ; stripped COERCEND.
b) called MODE FORM : fitted procedure delivering a MODE FORM.
c) called FORM : fitted procedure FORM.
d) stripped COERCEND : peeled procedure delivering a COERCEND.
e) peeled COERCEND : COERCEND ; stripped COERCEND.

{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: It is preelaborated {1.1.6.f} and a copy is made of {the routine which is} the resulting value ;
Step 2: The unaccompanied-call is replaced by the copy obtained in Step 1 and the elaboration of the copy is initiated; its value, if any, is then that of the unaccompanied call; 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 coercends

8.2.2.1. Syntax

a) expressed procedure delivering a COERCEND :
COERCEND ; hip expressed COERCEND ; depressed COERCEND.
b) expressed procedure cohesion : cohesive statement.

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

{For cohesive-statements see 6.2.1.d. }

8.2.2.2. Semantics

An expressed-coercend is elaborated in the following steps:
Step 1: A copy is made of it {itself, not its value} ;

.2.2.2. continued

tep 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-coercend; its mode is that obtained by deleting 'expressed' and the terminal production of 'FORM' from that notion as terminal production of which the expressed-coercend is elaborated.

{If *e1*, *e2* and *e3* are label-identifiers, then the reader might ecognise the effect of the declaration [] *proc switch = (e1, e2, e3)* nd the unitary-statement *switch[i]* ; owever, the declaration [*1:3*] *proc switch(e1, e2, e3)* s perhaps more powerful, since the assignation *switch*[2] := *e1* s possible.

The elaboration of *real expr(p | x | -x)* yields the routine *((p | x | -x))*, whereas that of the expressed-coercend *(p | x | -x)* yields either *(x)* or the routine *(-x)*, depending on the value of *p*. Similarly, the elaboration of *real expr(x := x + 1 ; y)* yields the routine *((x := x + 1 ; y))*, whereas that of the expressed-coercend *(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 e.g. a formula (8.1) or cohesion (8.3.1.b), then the elaboration of *real expr C* and that of the expressed-coercend *C* both yield the routine *(C)*. }

8.2.3. Depressed coercends

{"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 COERCEND : fitted reference to COERCEND.

{Example:
a) *x* (in *x* ↑ *2*) }

8.2.3.2. Semantics

A depressed-coercend is elaborated in the following steps:
Step 1: It is preelaborated {1.1.6.f} ;
Step 2: The value of the depressed-coercend is a new instance of the value referred to by the name obtained in Step 1.

8.2.4.1. Syntax

a) united union of LMODESETY MODE RMODESETY mode FORM : adjusted MODE FORM.

{Examples:
a) *one* (in *f + one*, see 11.11.bb) ; *b* (in *b + x*, ibid.) ; *call (fdash, g)* (in 11.11.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 coercends

8.2.5.1. Syntax

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

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

8.2.5.2. Semantics

A widened-coercend is elaborated in the following steps:
Step 1: It is preelaborated {1.1.6.f} ;
Step 2: If the value yielded by Step 1 is an integer, then the value of the widened-coercend is a new instance of that real number which is equivalent to that integer {2.2.3.1.d}; otherwise, it is a new instance of that structured {complex (10.2.5)} value composed of two fields, whose field-selectors are letter-r-letter-e and letter-i-letter-m, whose modes are the same as that of the value yielded in Step 1 and which are new instances of that value and zero respectively; its mode is that obtained by deleting 'widened' and the terminal production of 'FORM' from that notion as terminal production of which the widened-coercend is elaborated.

.6. Arrayed coercends

.6.1. Syntax

arrayed REFETY row of MODE confrontation :
  adapted REFETY MODE confrontation.
arrayed REFETY row of MODE ADIC formula :
  adapted REFETY MODE ADIC formula.
arrayed REFETY row of MODE cohesion : adapted REFETY MODE cohesion option.

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

.6.2. Semantics

arrayed-coercend is elaborated in the following steps:
p 1: If it is not empty, then it is preelaborated, and Step 3 is
aken ;
p 2: A new instance of a multiple value {2.2.3.3} composed of zero
lements and a descriptor consisting of an offset 1 and one quintuple
1, 0, 1, 1, 1) is considered, and Step 6 is taken ;
p 3: If the value obtained in Step 1 is a name, then the value
eferred to by this name, and, otherwise, the value itself obtained
n Step 1 is considered; if the considered value is a multiple value,
hen Step 5 is taken ;
p 4: A new instance of a multiple value composed of the considered
alue as only element, and a descriptor consisting of an offset 1
nd one quintuple (1, 1, 1, 1, 1) is considered instead, and Step 6
s taken ;
p 5: A new instance of a multiple value is created, composed of
he elements of the considered value and a descriptor which is a
opy of the descriptor of the considered value into which the
dditional quintuple (1, 1, 1, 1, 1) {the value of the stride is
rrelevant} is inserted before the first quintuple, and in which
ll states have been set to 1, and this new multiple value is
onsidered instead ;

8.2.6.2. continued

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

8.3. Primaries

8.3.1. Syntax

a) COERCETY MODE primary :
  COERCETY CLOSED MODE expression ; COERCETY MODE cohesion.
b) MODE cohesion : MODE denotation ; MODE identifier ; MODE slice ;
  nonlocal MODE generator ; MODE named TAG selection ; MODE expression call
c) hip FORCED MODE cohesion :
  FORCED MODE cohesion ; MODE hop ; MODE nihil.
d) NONPROC hop : skip ; jump.
e) reference to MODE nihil : nil symbol.

{Examples:
a) *(a | b | false) ; sin(b - a) ;*
b) *true ; x ; x2[i, j] ; compl(1, 0) ; father of algol ; sin(b - a) ;*
c) *x (in y := x) ; skip ; nil ;*
d) *skip ; goto grenoble ;*
e) *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, for expression-calls see 8.7, for skips see 6.2.1.e and for jumps see 6.2.1.f. }

3.2. Semantics

identifier is elaborated by considering a new instance of
~~The value of an identifier is~~ the value, if any, possessed by its
fining occurrence {4.1.2, 7.4.2. Step 5}; its value is then the
nsidered value.
{The identifier *pi* as declared in the standard declaration 10.3.a
a real-identifier (and not a reference-to-real-identifier). Its
lue cannot be changed by assignment. In fact, in this context,
:= 3 is not a production of 'assignation' (8.8). Similarly, the
entifier *sin* as declared in 10.3.g is a procedure-with-a-real-
rameter-delivering-a-real-identifier (5.4.1.b) and
n := *(real x) real : (x - x ↑ 3/6)* is also not an assignation. The
itialised declaration *real ppi(pi)* creates a name possessed by the
eference-to-real-identifier *ppi*, which name refers to the value of
; moreover, another value may be assigned to that name. }

) The value of a skip is a new instance of some value whose mode is
hat obtained in the following steps:
tep 1: The mode obtained by deleting 'hop' from that notion ending
with 'hop' of which the skip is a terminal production is considered ;
tep 2: If the considered mode begins with 'union of' then some mode
which does not begin with 'union of' and from which the considered
mode is united {2.2.4.1.h} is considered instead; the considered
mode is the mode of the value of the skip.

) A jump {see also 6.2.2.b} does not possess a value.

) The elaboration of a nihil involves no action; its value is a new
nstance of nil {2.2.3.5.a} whose mode is that obtained by deleting
nihil' from that notion ending with 'nihil' of which the nihil is a
erminal production.

{Skips play a role in the Semantics of routine-denotations (5.4.2.ii),
ormulas (8.1.2. Step 3) and accompanied-calls (8.7.2. Step 4).
oreover, they are useful in a number of programming situations, like e.g.
) Supplying an actual-parameter (7.4.1.b) or ~~structure~~ void (8.5.1.g)
whose value is irrelevant or is to be calculated later; e.g.
f(3, skip) where *f* does not use its second actual-parameter if the
value of its first actual-parameter is positive; see also 11.11.ax ;
ii) Supplying a constituent unitary-expression of a collateral-expression,
e.g. *[1:] real x1(3.14, skip, 1.68, skip)* ;

8.3.2. continued

iii) as a "dummy" statement (6.2.1.d) in those rare situations where the
use of a completer is inappropriate, e.g. *l : skip)* in 10.4.a ;
A jump is useful as expression to terminate the elaboration of another
expression when certain requirements are not met, e.g.
go to exit in *y := if x ≥ 0 then sqrt(x) else goto exit fi*,
f in *(j > a | f | j)* from 10.2.3.r.
A nihil is useful particularly where structured values are connected to
one another in that a field of each structured value refers to another
one except for one or more structured values where that field does not
refer to anything at all; the value of such a field must then be nil, e.g
language(14, nil) in the examples of 8.6.1.
Since the value of a nihil, nil, is unique, an identity-relation, like
father of father of algol :=: nil
can be used to determine whether a field is nil or not. }

8.4. Slices

8.4.1. Syntax

a) REFETY ROWSETY ROWWSETY NONROW slice : REFETY ROWS ROWSETY NONROW who
sub symbol, ROWS leaving ROWSETY indexer, bus symbol.
b) NONREF whole : NONREF primary ; called NONREF primary.
c) reference to NONREF whole : fitted reference to NONREF primary.
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 : hip fitted integral formary.
k) subscript : hip fitted integral formary.
l)* trimscript : trimmer option ; subscript.
m)* indexer : ROWS leaving ROWSETY indexer.

8.4.1. continued

{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* ;

h) *2:n:1* ;

i) *:1* ;

j) *1* ;

k) *i* }

{For primaries see 8.3 and for hip-fitted-formaries see 8.0.1.b. }

{In rule a, 'ROWS' reflects the number of trimscripts in the slice, 'ROWSETY' 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, *2:3:1* ; *2:n* ; *2:* ; *:5* and *::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: Its constituent whole, and all constituent subscripts, strict-lower-bounds, strict-upper-bounds and new-lower-bounds of the constituent indexer of the slice 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 considered, a copy is made of its descriptor, and all the states {2.2.3.3.b} in the copy are set to 1 ;

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

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

8.4.2. continued

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 $l_i \le l$ and $u \le u_i$, then the offset in the copy is increased by $(l - l_i) \times d_i$, and then l_i is replaced by l' and u_i by $(l' - l) + u$;

otherwise, the further elaboration is undefined ;

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

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

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 a new instance of the considered value itself.

{A trimmer restricts the possible values of a subscript and changes its notation: first, the value of the subscript is restricted to run from the value of the strict-lower-bound 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*. }

. Generators

.1. Syntax

{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.}

generator : local MODE generator ; nonlocal MODE generator.
local MODE generator : local symbol, nonlocal MODE generator.
nonlocal reference to MODE generator :
  actual MODE declarer, MODE initialisation option.
MODE initialisation :
  hip adapted CLOSED MODE expression ; MODE structure pack.
structured with FIELDS and a FIELD structure :
  structured with FIELDS structure, comma symbol,
  structured with a FIELD structure.
structured with a MODE named TAG structure : MODE veld.
MODE veld:
  hip adapted unitary MODE expression ; MODE structure pack.

{Examples:
loc[1 : 3] *real* (1.2, 3.4, 5.6) ;
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*)) ;
(z) ; (1, 0) ;
1, 0 ;
1 ; (2.3, *true*) }

.2. Semantics

A given structure is elaborated in the following steps:
ep 1: All constituent ~~expressions and structures~~ velds of the given structure
re elaborated collatorally {6.3.2.a} ~~;~~, where the elaboration of a
ructure-pack is that of its constituent structure.
ep 2: The values obtained in Step 1 are made, in the given order,
o be the fields of a new instance of a structured value {2.2.3.2} whose
ode is obtained by deleting 'structure' from that notion ending with
structure' of which the given structure is a terminal production, and

8.5.2. continued

this structured value is the value of the given structure.

b) A generator is elaborated in the following steps:
Step 1: Its constituent actual-declarer {7.1.2.c} and initialisation, if any, are elaborated collaterally~~; the elaboration of a structure-pack being that of its constituent structure~~ ;
Step 2: If there is an initialisation, then its value is assigned {8.8.2.a} to the value {name} of the actual-declarer ;
Step 3: The value of the generator is the value {name} of the actual-declarer.

c) The scope {2.2.4.2} of the value of a local-generator is the smalles 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~~ replaced by
  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~~ it violates the condition on scopes (8.8.2.c. 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).~~ trimscript (see 8.4.2. Step 4, 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.c). }

8.6. Field selections

8.6.1. Syntax

a) REFETY FIELD selection : FIELD selector, of symbol,
REFETY structured with LFIELDSA FIELD RFIELDSETY whole.

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

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

{Rule a 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 is the same sequence of symbols as a field-selector in the same range creates no ambiguity.

8.6.1. continued

Thus *age of algol := age* is a (possibly 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-letter-g-letter-e' 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 PARAMETERS delivering a MODE primary,
actual PARAMETERS pack.
c) statement call : fitted procedure with PARAMETERS primary,
actual PARAMETERS pack.

{Examples:
b) *samelson(m, (int j) real : x1[j])* (in the scope of
proc samelson = (int n, proc(int) real f) real :
begin long real s(long 0) ;
for i to n do s plus leng f(i) ↑ 2 ;
short long sqrt(s) end) ;
c) *set random(x)* ; (see 10.3.2) }

{For actual-parameters see 7.4.1.b and for fitted-primaries see 8.3.1.a. See also unaccompanied-calls, 8.2.1. }

8.7.2. Semantics

An accompanied-call is elaborated in the following steps:
Step 1: Its constituent fitted-primary is elaborated and a copy is made of {the routine which is} its value ;
Step 2: The accompanied-call is replaced by that copy ;

8.7.2. continued

Step 3: That copy {which is now a closed-clause} is protected {6.0.2.d} ;
Step 4: The copy as modified by Step 3 is further modified by replacing the skip-symbols following the equals-symbol following the constituent formal-parameters of the copy {5.4.2.ii} in the textual order by the constituent actual-parameters of the accompanied-call taken in the same order ;
Step 5: The elaboration of the copy is initiated; its value, if any, is then that of the accompanied call; 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) real : 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 plus leng f(i) ↑ 2 ;
short long sqrt(s) end).

Supposing that *n*, *f*, *s* and *i* do not occur elsewhere, this closed-expression is protected (Step 3) without further alteration. The actual-parameters are now inserted (Step 4) yielding the closed-expression

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

and this closed-expression is elaborated (Step 5). Note that, for the duration of this elaboration, *n* possesses the same integer as that referred to by the name possessed by *m*, and *f* possesses the same routine as that possessed by the routine-denotation *(int j) real : x1[j]*. During the elaboration of this and its inner nested closed-clauses (9.3), the elaboration of *f(i)* itself involves the elaboration of the closed-expression *((int j = i) ; x1[j])*, and, within this inner closed-expression, the first occurrence of *j* possesses 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 formary.

c) MODE source : hip 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 }

{For peeled-formaries see 8.0.1.b and for hip-adapted-unitary-expressions see 8.0.1.a }

8.8.2. Semantics

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

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

c) A value is assigned to a name in the following steps:

Step 1: If the given value does not refer to an element or subvalue of a multiple value having one or more states equal to zero {2.2.3.3.b}, 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: The value referred to by the given name is considered; if the mode of the given name does not begin with 'reference to union of' and the considered value is a multiple value or a structured value, then Step 3 is taken; otherwise, the considered value is superseded by a new instance of the given value and the assignment ~~is complete ;~~ has been accomplished;

8.8.2. continued

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

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

Step 5: Each element (field) of the ~~considered~~ given value is ~~superseded by a new instance of~~ assigned {in an order which is left undefined} to the name referring to the corresponding element (field) of the ~~given~~ considered value and the assignment ~~is complete. {The order in which these elements (fields) are superseded is undefined.}~~ has been accomplished

d) 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 a new instance of 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'. }

8.9. Conformity relations

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

8.9.1. Syntax

a) boolean conformity relation :
peeled reference to LMODE formary, conformity relator, RMODE formary.

b) conformity relator :
conforms to symbol ; conforms to and becomes symbol.

9.1. continued

{Examples:

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

:: ; ::= }

{For peeled-formaries and formaries see 8.0.1.b. }

9.2. Semantics

conformity-relation is elaborated in the following steps:

ep 1: Its constituent peeled-formary and formary are elaborated collaterally {6.3.2.a} and the value of that formary is considered ;

ep 2: If the mode of the value of the peeled-formary is 'reference to' followed by a mode which is or is united from {2.2.4.1.h} the mode of the considered value, then the value of the conformity-relation is true and Step 4 is taken; otherwise, Step 3 is taken ;

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

ep 4: If the constituent conformity-relator is a conforms-to-and-becomes-symbol and the value of the conformity-relation is true, then the considered value is assigned {8.8.2.c} to the value of the peeled-formary.

{Observe that if the considered value is an integer and the mode of the alue of the peeled-formary is 'reference to' followed by a mode which is is united from the mode 'real' but not from 'integral', then the value f the conformity-relation is false. Thus, in contrast with the assignation, o automatic widening from integral to real takes place. }

.10. Identity relations

.10.1. Syntax

) boolean identity relation : peeled reference to MODE formary, identity relator, hip peeled reference to MODE formary.

) identity relator : is symbol ; is not symbol.

.10.1. continued

{Examples:

) *xx* :=: *yy* ; *val xx* :=: *x or y* ;

) :=: ; :≠: }

{For peeled-formaries and hip-peeled-formaries see 8.0.1.b. }

.10.2. Semantics

n identity-relation is elaborated in the following steps:

Step 1: Its constituent peeled-formary and hip-peeled-formary are elabora collaterally {6.3.2.a} ;

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

{Assuming the assignation *xx* := *yy* := *x*, the value of the identity-relation *xx* :=: *yy* is false because *xx* and *yy*, though of the same mode, do not possess the same name. 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 possessed by *x*, and the routine possessed by *x or y* (see 1.3), when elaborated, yields either the name possessed by *x* or, with equal probability, the name possessed by *y*. In the identity relation, the programmer is usually asking a specific question concerning names and thus the level of reference is of crucial importance. Since no automatic depressing of the formaries 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 b 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 formaries are not names. Also,

f2d3df :=: *f5df*

is not an identity-relation, whereas

f2d3df = *f5df*

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

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~~ restrictions, 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~~ given in the representation language, except that

A stands for a unitary-expression {Chapter 8},
B for a unitary-expression,
C_1 and C_2 for unitary-clauses {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 two or more unitary-clauses separated by comma-symbols,
H for a declarer {7.1},
I, *J*, *K* and *L* for identifiers {4.1},
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 a tail {7.1.1.w, x, z},
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, and
Z for a formal-declarer {7.1.1.b} all of whose formal-row-of-~~rower-options~~ towers {7.1.1.ϕ} are empty.

d) Each representation of a symbol appearing in Sections 9.1 up to 9.5 may be replaced by any other representation, if any, of the same symbol.

9.1. Comments

{A source of innocent merriment.
Mikado, W.S. Gilbert.}

A comment {3.0.9.b} may be inserted between any two symbols {but see 9.b.}.

{e.g. *(m > n | m | n)* may be ~~written~~ replaced by
(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~~ replaced by *real x* and *ref bool p* = *loc bool(true)* may be ~~written~~ replaced by *bool p(true)*. }

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

{e.g. *mode compl* = *struct(real re, im)* (see also 9.2.c) may be ~~written~~ replaced by *struct compl* = *(real re, im)*. }

c) If a given unitary-declaration (formal-parameter {5.4.1.e}, field-declarator {7.1.1.g}) and another unitary-declaration (formal-parameter, field-declarator) following a comma-symbol following the given unitary-declaration (formal-parameter, field-declarator) both begin with an occurrence of the mode-symbol, of the structure-symbol, of the union-of-symbol, of the priority-symbol, of the operation-symbol, of one same actual-declarer, or of one same formal-declarer, then the second of these occurrences may be omitted.

{e.g. *real x, real y(1.2)* may be ~~written~~ replaced by *real x, y(1.2)*, but *real x, real y = 1.2* may not be ~~written~~ replaced by *real x, y = 1.2*, since the first occurrence of *real* is an actual-declarer whereas the second is a formal-declarer.

Note also that *mode b = bool, mode r = real* may be ~~written~~ replaced by *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 9.2.c} begins with an occurrence of a mode-symbol, structure-symbol, union-of-symbol, priority-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~~ replaced by *real x, y, z*. }

9.2. continued

e) *proc* *PI* = *R* may be replaced by *proc* *I* = *R*.

f) *op* *PN* = *R* may be replaced by *op* *N* = *R*.

g) *proc* *PO(R)* may be replaced by *proc* *O(R)*.

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

9.3. Repetitive statements

a) The unitary-statement

begin(int J(F), int K = B, L = T) ;
M : if D(K > O | J ≤ L |: K < O | 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 *D*, *W* or *S*, and where *I* differs from *J* and *K*, may be replaced by *for I from F by B to T while W do S* , and if, moreover, *I* does not occur in *W* or *S*, then *for I from* may be replaced by *from*.

b) The unitary-statement

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

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

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

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

{e.g. *for i from 1 by 1 to n while true do x := x + a* may be ~~written~~ replaced by *to n do x := x + a*.

9.3. continued

Note that *to 0 do S* and *while false do S* do not cause *S* to be elaborated at all, whereas *do S* causes *S* to be elaborated repeatedly until it is terminated or interrupted. }

9.4. Contracted conditional clauses {The flowers that bloom in the spring,
Tra la,
Have nothing to do with the case.
Mikado, W.S. Gilbert.

a) *else if Q fi fi* may be replaced by *elsf Q fi* and
then if Q fi fi may be replaced by *thef Q fi*.

{e.g. *if p then princeton else if q then grenoble else zandvoort fi fi* may be ~~written~~ replaced by

if p then princeton
elsf q then grenoble else zandvoort fi

or *(p | princeton |: q | grenoble | zandvoort)*.
Many more examples are given in 10.5. }

b) *(int I = A ; if DI = 1 then C_1 elsf D(I = 2 | true) then C_2 fi)*, where *I* does not occur in C_1, C_2 or *D*, may be replaced by
case A in C_1, C_2 esac.

c) *(int I = A ; if DI = 1 then C_1 else case(DI - 1) in G esac fi)*, where *I* does not occur in C_1, *D* or *G*, may be replaced by
case A in C_1, G esac.

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

9.5. Complex values

val(L real I = A, J = B ; (DL compl(I, J))),
where *I* and *J* do not occur in *D*, may be replaced by
(A ⊥ B).

10. Standard declarations

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

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

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

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

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

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

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

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

10. continued

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

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

Step 7: Each occurrence of *F* in any form, as possibly modified or made in the Steps above, is replaced by a representation of one same terminal production of a terminal production of 'ALEPH' {1.2.5.d, 5.5.1.6.a} ;

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

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

Step 10: If, in any form, as possibly modified or made in the Steps above, a representation of a routine-denotation occurs whose elaboration involves the manipulation of real numbers, then this denotation may be replaced by any other denotation whose elaboration has approximately the same effect {The degree of approximation is ~~not defined~~ left undefined in this Report (see also 2.2.3.1.c).} ;

Step 11: The standard-declarations are that serial-declaration followed by a go-on-symbol whose representation is the same as the sequence of all the forms, as possibly modified or made in the Steps above.

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

10.1. Environment enquiries

a) *int int lengths = c the number of different lengths of integers c ;*
b) *L int L max int = c the largest L integral value c ;*
c) *int real lengths =*
c the number of different lengths of real numbers c ;
d) *L real L max real = c the largest L real value c ;*
e) *L real L small real = c the smallest L real value such that both*
L1 + L small real > L1 and L1 - L small real < L1 c ;
f) *int bits widths =*
c the number of different widths of standard bit rows c ;
g) *int L bits width =* ~~{10.2.6.a}~~ ; see L bits
c the number of bits in a standard L bit row ~~c~~ {10.2.6.a} *c ;*
h) *op abs = (char a) int :*
c the integral equivalent of the character 'a' c ;
i) *op repr = (int a) char :*
c that character 'x', if it exists, for which abs x = a c ;

10.2. Standard priorities and operations

10.2.0. Standard priorities

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

10.2.1. Operations on boolean operands

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

10.2.2. Operations on integral operands

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

10.2.2. continued

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

10.2.3. Operations on real operands

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

10.2.3. continued

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

10.2.4. Operations on arithmetic operands

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

10.2.5. Complex structures and associated operations

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

10.2.5. continued

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

10.2.6. Bit rows and associated operations

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

10.2.7. Operations on character operands

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

10.2.8. String mode and associated operations

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

a R string(b)

string(a) R b

{The operation defined in b implies that if "a"[1] < "b"[1], then "" < "a" ; "a" < "b" ; "aa" < "ab" ; "aa" < "ba" ; "ab" < "b". }

10.2.9. Operations combined with assignations

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

10.2.9. continued

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

10.3. Standard mathematical constants and functions

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

10.3. continued

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

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

10.4. Synchronization operations

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

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

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

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

10.5.0. Transput modes and straightening

10.5.0.1. Transput modes

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

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

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

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

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

10.5.0.2. Straightening

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

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

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

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

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

10.5.0.2. continued

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

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

10.5.1. Channels and files

{"Channels", "backfiles" and files model the transput devices of the physical machine used in the implementation.
A channel corresponds to a device, e.g. a card reader or punch, a magnetic drum or disc, a piece of core memory, a tape unit or even a set-up in nuclear physics the results of which are collected by the computer. A channel has certain properties (10.5.1.1.d : 10.5.1.1.m). Some devices may be seen as channels with properties in more than one way. The choice mode in an implementation is a matter for individual taste. Some possible choices are given in Table I.

All information on a given channel is to be found in a number of backfiles. A backfile (10.5.1.1.b) comprises ~~a reference to~~ a three-dimensional array of integers (bytes of information), the *book* of the backfile, indexed by *page*, *line* and *char*; the lower bounds of the *book* are all one and the upper bounds are the *maxpage*, *maxline* and *maxchar* of the channel; furthermore, the backfile comprises the position of the "end of file", i.e. the page number, line number and character number up to which the backfile is filled with information.
On a given channel, a certain maximum number (10.5.1.1.m) of files (10.5.1.2.a) may be "opened" at any stage of the elaboration of the program. A file contains a reference to a backfile, to a current page number, line number and character number, and to the channel on which it has been opened.

After the elaboration of the declaration of *nextbfile* (10.5.1.1.c), all backfiles are part of the chains of backfiles referred to by *nextbfile*.

10.5.1. continued

Examples:

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

ii) Channel four is a drum, divided into 32 segments each being one page of 256 lines of 256 bytes. It has 32 backfiles of input information (the previous contents of the drum), so *nextbfile*[4] is initialized to refer to the first ~~element~~ backfile of a chain of 32 backfiles, each referencing the next, the last one having *next* equal to nil. Each of those backfiles has an end of file at position (2, 1, 1).

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

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

In opening (10.5.1.2.c) a given file on a given channel, the first backfile is taken from the chain referenced by *nextbfile* of the channel and is made to be referred to by *bfile* of the file, obliterating the previous backfile, if any, of the file.

Apart from the possibility of being obliterated, at any stage in the elaboration of the particular-program, all backfiles are either part of the *nextbfile* chain of the channel, or referenced by a file opened on that channel, or part of the *closedbfile* chain (10.5.1.1.o) of that channel.

This models the ~~part~~ fate of a magnetic tape that, apart from the possibility of being left as a scratch tape, is either ready to be mounted on, in use on, or saved from the tape unit.

When a file is "closed" {10.5.1.2.q}, its backfile is attached to the chain referenced by *closedbfile* of the channel; all files referencing the same backfile as that file are then unavailable for further transput.

Example:

```
begin  file tape1, tape2 ;
       open(tape1, 20) ; tape2 := tape1 ;
       out(tape2, fd.2df, pi) ; new page(tape1) ;
       close(tape2)
end
```

The *conv* of a given file is used in conversion; if *conv* of the file is nil, then *stand conv* of the channel on which the file was opened is used as "conversion key", and, otherwise, the string to which *conv* of the file refers.

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

On input, if an integer to be converted is smaller than one or larger than the number of elements of the conversion key, then the further elaboration is undefined; otherwise, the integer is converted to that character in the key whose ordinal number is that integer.

On all files opened on a channel for which *set possible* is false, and *put possible* and *get possible* both true, input and output may not be "alternated", i.e. after opening or resetting {10.5.1.2.e} such a file, either is possible, but, once one has taken place on the file, the other may not until the file has been reset again. Before the first output takes place on such a file, its *book* is filled with spaces. If, after output, such a file is reset, then an end of file is positioned at the current page, line and character number. (Such a file might be implemented with a buffer that holds one line.)

On all files opened on a channel for which *set possible* is false, binary and nonbinary transput may not be alternated. }

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

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

TABLE I: Properties of some possible channels

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

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

TABLE I: Properties of some possible channels

10.5.1.1. Channels

a) int nmb channels = c an integral-expression indicating the number of transput devices in the implementation c ;

b) struct % bfile = ([,,] int book, int lpage, lline, lchar, ref bfile next) ;

c) [1 : nmb channels] ref bfile % nextbfile(c some appropriate initialization c) ;

d) [1 : nmb channels] bool reset possible = c a row-of-boolean-expression, indicating which of the physical devices corresponding to the channels allow resetting {e.g. rewinding of a magnetic tape} c ;

e) [1 : nmb channels] bool set possible = c a row-of-boolean-expression, indicating which devices can be accessed at random c ;

f) [1 : nmb channels] bool get possible = c a row-of-boolean-expression indicating which devices can be used for input c ;

g) [1 : nmb channels] bool put possible = c a row-of-boolean-expression indicating which devices can be used for output c ;

h) [1 : nmb channels] bool bin possible = c a row-of-boolean-expression indicating which devices can be used for binary transput c ;

i) [1 : nmb channels] int max page = c a row-of-integral-expression giving the maximum number of pages per file for the channels c ;

j) [1 : nmb channels] int max line = c a row-of-integral-expression giving the maximum number of lines per page c ;

k) [1 : nmb channels] int max char = c a row-of-integral-expression giving the maximum number of characters per line c ;

l) [1 : nmb channels] ref string % stand conv = c a row-of-reference-to-row-of-character-expression giving the standard conversion keys for the channels c ;

m) [1 : nmb channels] int max nmb files = c a row-of-integral-expression giving the maximum numbers of files the channels can accommodate c ;

n) [1 : nmb channels] int % nmb opened files
([1 : nmb channels] int zero ; for i to nmb channels do zero[i] := 0 ; zero) ;

o) [1 : nmb channels] ref bfile % closedbfile
([1 : nmb channels] ref bfile nil ; for i to nmb channels do nil[i] := nil ; nil) ;

10.5.1.1. continued

p) int stand in channel = c an integral-expression whose value does not exceed nmb channels, such that get possible [stand in channel] is true, and stand conv [stand in channel] comprises, in some order, all character-tokens c ;

q) int stand out channel = c an integral-expression whose value does not exceed nmb channels, such that put possible [stand out channel] is true, and stand conv [stand out channel] comprises, in some order, all character-tokens c ;

r) int stand back channel = c an integral-expression whose value does not exceed nmb channels, such that reset possible [stand back channel] set possible [stand back channel], get possible [stand back channel], put possible [stand back channel] and bin possible [stand back channel] are true and stand conv [stand back channel] comprises, in some order, all character tokens c ;

s) proc file available = (int channel) bool :
(nmb opened files [channel] < max nmb files [channel]) ;

10.5.1.2. Files

```
a) struct file = (ref bfile % bfile,
                  ref int % page, % line, % char, % chan,
                  ref bool % state def, % state get, % state bin, % opened,
                  ref string conv) ;
b) proc % undefined = expr((false | true) | skip) ;
c) proc open = (ref file file, int ch) :
     if file available (ch)
     then file := file(nextbfile[ch], 1, 1, 1, ch, false, false, false,
          true, nil) ; nextbfile[ch] := next of nextbfile[ch] ;
          nmb opened files[ch] plus 1
     else undefined
     fi ;
d) proc set = (file file, int p, l, c) :
     if set possible[chan of file] ∧ opened of file
     then page of file := p ; line of file := l ; char of file := c ;
          (outside(file) | undefined)
     else undefined
     fi ;
```

```
e) proc reset = (file file) :
     if reset possible[chan of file] ∧ opened of file
     then if state def of file ∧ ¬ state get of file ∧
     ¬ set possible[chan of file]
          then lpage of bfile of file := page of file ;
               lline of bfile of file := line of file ;
               lchar of bfile of file := char of file
          fi ;
          page of file := line of file := char of file := 1 ;
          state def of file := false
     else undefined
     fi ;
f) proc % outside = (file file) bool : (opened of file |
     line ended(file) ∨ page ended(file) ∨ file ended(file)) ;
g) proc file ended = (file file) bool : (opened of file |
     int p = page of file, lp = lpage of bfile of file,
         l = line of file, ll = lline of bfile of file,
         c = char of file, lc = lchar of bfile of file ;
         (p < lp | false |: p > lp | true |: l < ll | false
          |: l > ll | true | c ≥ lc)) ;
h) proc line ended = (file file) bool :
     (opened of file | char of file > max char[chan of file]) ;
i) proc page ended = (file file) bool :
     (opened of file | line of file > max line[chan of file]) ;
j) proc % get string = (file file, ref[1 : int n] char s) :
     if get possible[chan of file] ∧ opened of file
     then string conv = (conv of file :=: nil | stand conv[chan of file] |
          conv of file) ;
          int p = page of file, l = line of file ; ref int c = char of file
          if ¬ set possible[chan of file] thef state def of file
          then(¬state get of file ∨ state bin of file | undefined)
          else state def of file := state get of file := true ;
               state bin of file := false
          fi ;
          for i to n do(outside(file) | undefined |
               s[i] := conv[book of bfile of file[p, l, c]] ; c plus 1)
     else undefined
     fi ;
```

char of file < 1 ∨
line of file < 1 ∨
page of file < 1 ∨


```
k) proc % put string = (file file,[1 : int n] char s) :
     if put possible[chan of file] ∧ opened of file
     then int ch = chan of file, p = page of file, l = line of file ;
          string conv = (conv of file :=: nil | stand conv[ch] |
               conv of file) ; int space, h ; ref int c = char of file ;
          if ¬ set possible[ch] thef state def of file
          then(state get of file ∨ state bin of file | undefined)
          else state def of file := true ; state get of file :=
               state bin of file := false ;
               (¬ char in string(".", space, conv) | undefined) ;
               for i to max page[ch] do for j to max line[ch] do
               for k to max char[ch] do
                    book of bfile of file[i, j, k] := space
          fi ;
          for i to n do(¬ char in string(s[i], h, conv) ∨ outside(file) |
               undefined | book of bfile of file[p, l, c] := h ; c plus 1
     else undefined
     fi ;
l) proc char in string = ([1 : 1] char c, ref int i, [1 : int w]char s)boo
     (for k to w do(c = s[k] | i := k ; 1) ; false. 1 : true) ;
m) proc space = (file file) :
     (char of file plus 1 ; outside(file) | undefined) ;
n) proc backspace = (file file) :
     (char of file minus 1 ; outside(file) | undefined) ;
o) proc new line = (file file) :
     (line of file plus 1 ; char of file := 1 ; outside(file) | undefined)
p) proc new page = (file file) :
     (page of file plus 1 ; line of file := char of file := 1 ;
      outside(file) | undefined) ;
q) proc close = (file file) :
     if opened of file
     then int ch = chan of file ; (reset possible[ch] | reset(file)) ;
          next of bfile of file := closedbfile[ch] ;
          closedbfile[ch] := bfile of file ; opened of file := false ;
          rmb opened files[ch] minus 1
     fi ;
```

```
r) file stand in = (file f ; open(f, stand in channel) ; f) ;
s) file stand out = (file f ; open(f, stand out channel) ; f) ;
t) file stand back = (file f ; open(f, stand back channel) ; f) ;
```

{Certain "standard files" (r, s, t) need not be opened by the programmer, but are opened for him in the standard-declarations; ~~The procedure~~ *print* (10.5.2.a) can be used for output on *stand out*, *read* (10.5.3.a) for input from *stand in*, and *write bin* {10.5.6.a} and *read bin* {10.5.7.a} for transput involving *stand back*. }

```
u) proc char number = (file f) int : (opened of f | char of f) ;
v) proc line number = (file f) int : (opened of f | line of f) ;
w) proc page number = (file f) int : (opened of f | page of f) ;
```

10.5.2. Formatless output

```
a) proc print = ([] outtype x) : put(standout, x) ;
b) proc put = (file file, [1 : int n] outtype x) : (for i to n do
    ([1 : int l] simplout y = straightout x[i] ;
     for j to l do
     (string s ; bool b ; char c ;
     (↓ (L int i ; (i ::= y[j] |
        s := L int string(i, L int width + 1, 10) ;
        sign supp zero(s[1], L int width - 1))) ↓) ;
     (↓ (L real x ; (x ::= y[j] | s := L real string
        (x, L real width + L exp width + 4, L real width - 1, L exp width)
        sign supp zero(s[1:])) ↓) ;
     (↓ (L compl z ; (z ::= y[j] |
        put(file, (re z, "|.", im z)) ; end)) ↓) ;
     (b ::= y[j] | s := (b | "1" | "0")) ;
     (c ::= y[j] | nextplc(file) ; put string(file, c) ; end) ;
     (s ::= y[j] | ref[1 : int n] char t = s ;
        for i to n do put(file, s[i]) ; end) ;
     (ref[1 : int n] char t = s ; int c1 = char of file ;
      char of file := c1 + n ; (line ended(file) | nextplc(file) |
        char of file := c1) ; put string(file, s) ;
      (¬line ended(file) | space(file))) ;
    end: skip))) ;
```



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

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

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

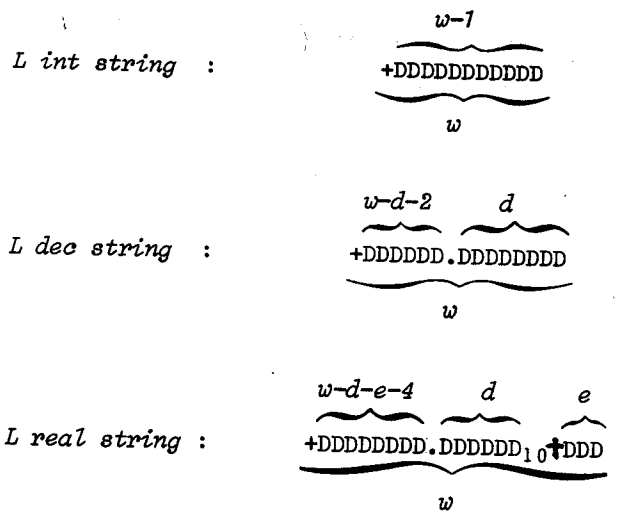

TABLE II: Display of the values of
L int string, *L dec string* and *L real string*

frame

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

TABLE III: Significance of the elements of *frame*

10.5.3. Formatless Input

```
a) proc read = ([] intype x) : get(stand in, x) ;
b) proc get = (file file, [1 : int w] intype x) :
     begin char k ;
       op ? = (string s) bool :
       (outside(file) | false |: get string(file, k) ;
         char in string(k, loc int, s) |
         true | backspace(file) ; false) ;
       proc read num = string expr
       (skip spaces ; ? "+ -" | k + (skip spaces ; read dig) |
                                "+" + read dig) ;
       proc skip spaces = expr while(nextplc(file) ; ? ".") do skip ;
       proc read dig = string expr
         (string t("") ; while ? "0123456789" do t plus k ; t) ;
       proc read real = string expr
         (read num + (? "." | "." + read dig | "") +
         (? "10" | "10" + (? "+" | k | "+") + read dig | "")) ;
       for i to w do
       ([1 : int l] ref simplout y = straightin x[i] ;
       for j to l do
       (ref bool bb ; ref string ss ; ref char cc ;
       (↓ (ref L int ii ; (ii ::= y[j] |
         val ii := L string int(read num, 10))) ↓) ;
       (↓ (ref L real xx ; (xx ::= y[j] |
         val xx := L string real(read real))) ↓) ;
       (↓ (ref L compl zz ; (zz ::= y[j] | get(file, re of zz) ;
         (skip spaces ; ? "|" | get(file, im of zz) | undefined))) ↓) ;
       (bb ::= y[j] | skip spaces ; val bb := (? "10" | k = "1")) ;
       (cc ::= y[j] | nextplc(file) ; get string(file, cc) ; end) ;
       (ss ::= y[j] | ref[1: int n] char t = ss ;
         for i to n do get(file, ss[i]) ; end) ;
       (¬line ended(file) |: ~~get(file, cc) ; cc ≠~~ "." | undefined) ;
     end: skip end ;
c) proc L string int = ([1 : int w] char x, int r) L int :
     (L int n(L0) ; L int lr = Kr ; for i from 2 to w do
     n := n × lr + K(int d = char dig(x[i]) ; (d < r | d)) ;
     (x[1] = "+" | n |: x[1] = "-" | -n)) ;
```

10.5.3. continued

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

10.5.4. Formatted output

```
a) proc out = (file file, tamrof tamrof, [1 : int n] outtype x) :
     begin string format = format primary list pack
           ("("+ F of tamrof +")", loc int(1)) ; int p(1) ;
           for k to n do
           ([1 : int l] simplout y = straightout x[k] ;
           for j to l do
           ([1 : 14] int frame ; int q(p) ; pattern(format, p, frame) ;
           (frame[1] | int, real, real, compl, compl, string, intch, bool)
       int: (↓ (L int i ; (i ::= y[j] |
           trans edit L int(file, i, format, q, frame) ; end)) ↓) ;
           undefined.
      real: (↓ (L real x ; (x ::= y[j] |
           trane edit L real(file, x, format, q, frame) ; end)) ↓) ;
           (↓ (L int i ; (i ::= y[j] |
           trans edit L real(file, i, format, q, frame) ; end)) ↓) ;
           undefined.
     compl: (↓ (L compl z ; (z ::= y[j] |
           trans edit L compl(file, z, format, q, frame) ; end)) ↓) ;
           (↓ (L real x ; (x ::= y[j] |
           trans edit L compl(file, x, format, q, frame) ; end)) ↓) ;
           (↓ (L int i ; (i ::= y[j] |
           trans edit L compl(file, i, format, q, frame) ; end)) ↓) ;
           undefined.
```

10.5.4. continued

```
      string: ([1 : frame[4]] char s ; char c ;
           (s ::= y[j] |
           trans edit string(file, s, format, q, frame) ; end) ;
           (c ::= y[j] |
           trans edit string(file, c, format, q, frame) ; end)) ;
           undefined.
       intch: (int i ; (i ::= y[j] |
           trans edit choice(file, i, format, q) ; end)) ; undefined.
        bool: (bool b ; (b ::= y[j] |
           trans edit bool(file, b, format, q) ; end)) ; undefined.
         end: do insertion(file, format, q) ; p plus 1
           ))
     end ;
b) proc % format primary list pack = (string s, ref int p) string :
     (string t(format primary(s, p)) ;
      while s[p] = "," do t plus "," + format primary(s, p) ; p plus 1 ; t)
c) proc % format primary = (string s, ref int p) string :
     (int n, q ; string f(p plus 1 ; insertion(s, p)) ;
      q := p ; replicator(s, p, n) ;
     (s[p] = "(" | string t = format primary list pack(s, p) ;
      to n do f plus t | p := q ; f plus pattern(s, p, loc[1 : 14] int)) ;
     f + insertion(s, p)) ;
d) proc % insertion = (string s, ref int p) string :
     (int q = p ; skip insertion(s, p) ; s[q : p - 1 : 1]) ;
e) proc % skip insertion = ([1 : int l] char s, ref int p) :
     while(p > l | false |: skip align(s, p) | true |
      skip lit(s, p)) do skip ;
f) proc % skip align = (string s, ref int p) bool :
     (int q = p ; replicator(s, p, loc int) ;
     (char in string(s[p], loc int, "x y p l k") |
      p plus 1 ; true | p := q ; false)) ;
g) proc % replicator = (string s, ref int p, n) :
     (string t("") ; while char in string
      (s[p], loc int, "0123456789") do(t plus s[p] ; p % plus 1);
      n := (t = "" | 1 | string int("+" + t, 10))) ;
```

```
h) proc % skip lit = (string s, ref int p) bool :
     (int q = p ; replicator(s, p, loc int) ;
      (s[p] = """" | while(s[p plus 1] = """" | s[p plus 1] = """" |
       true) do skip ; true | p := q ; false)) ;
i) proc % pattern = ([1 : int m] char format, ref int p,
          ref[] int frame) string :
     begin int n ; int p0 = p;
           op ? = (string s) bool :
           (skip insertion(format, p) ; p > m | false |
            int q = p ; replicator(format, p, n) ;
             (format[p] = "s" | p plus 1) ;
            (char in string(format[p], loc int, s) | true | p := q ; false)) ;
           proc intreal pattern = (ref[1 : 7] int frame) bool :
           ((num mould(frame[2 : 4 : 1]) | frame[1] := 1 ; 1) ;
            (? "." |: num mould(frame[3 : 5 : 1]) | frame[1] := 2 ; 1) ;
            (? "e" |: num mould(frame[5 : 7 : 1]) | frame[1] := 3 ; 1) ;
            false. 1 : true) ;
           proc num mould = (ref[1 : 3] int frame) bool :
           ((? "r" | frame[1] := n) ; (? "z" | frame[3] plus n) ;
            (? "+" | frame[2] := 1 |: ? "-" | frame[2] := 2) ;
            while ? "dz" do frame[3] plus n ;
            format[p] = "," ∨ format[p] = "i") ;
           proc string mould = (ref[] int frame) bool :
           (while ? "a" do frame[4] plus n ; format[p] = ",") ;
           for i to 14 do frame[i] := 0 ;
           (intreal pattern(frame[1 : 7]) | (? "i" | p plus 1 ;
            frame[1] plus 2 ; intreal pattern(frame[8 : 14 : 1])) ; end) ;
           (string mould(frame) | frame[1] := 6 ; end) ;
           (? "b" | frame[1] := 8 | p plus 1 ; frame[1] := 7) ;
           (format[p] = "(" |
            while ? "(," do skip lit(format, p) ; p plus 1) ;
      end: skip insertion(format, p); format[p0 : p-1]
     end ;
```



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

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



```
a) proc in = (file file, tamrof tamrof, [1 : int n] intype x) :
    begin string format =
          format primary list pack("("+ F of tamrof +")", loc int(1)) ;
          int p(1) ;
          for k to n do
          ([1 : int l] ref simplout y = straightin x[k] ;
          for j to l do
          ([1 : 14] int frame ; int q(p) ; pattern(format, p, frame) ;
          (frame[1] | int, real, real, compl, compl, string, intch, bool)
      int: (↓ (ref L int ii ; (ii ::= y[j] |
          trans indit L int(file, ii, format, q, frame) ; end)) ↑) ;
          undefined.
     real: (↓ (ref L real xx ; (xx ::= y[j] |
          trans indit L real(file, xx, format, q, frame) ; end)) ↑) ;
          undefined.
     compl: (↓ (ref L compl zz ; (zz ::= y[j] |
          trans indit L compl(file, zz, format, q, frame) ; end)) ↑) ;
          undefined.
   string: (ref string ss ; ref char cc ; [1 : frame[4]] char t ;
          trans indit string(file, t, format, q, frame) ;
          (ss ::= y[j] | val ss := t ; end |: cc ::= y[j] |
          val cc := t[1] ; end)) ;
          undefined.
    intch: (ref int ii ; (ii ::= y[j] |
          trans indit choice(file, ii, format, q) ; end)) ;
          undefined.
     bool: (ref bool bb ; (bb ::= y[j] |
          trans indit bool(file, bb, format, q) ; end)) ;
          undefined.
      end: req insertion(file, format, q) ; p plus 1))
    end ;
```

```
b) proc % trans indit L int =
     (file f, ref L int i, string format, ref int p, [] int fr) :
     (string t ; trans indit string(f, t, format, p, fr) ;
      i := L string int(t, 10)) ;
c) proc % trans indit L real =
     (file f, ref L real x, string format, ref int p, [] int fr) :
     (string t ; trans indit string(f, t, format, p, fr) ;
      x := L string real(t)) ;
d) proc % trans indit L compl =
     (file f, ref L compl z, string format, ref int p, [] int fr) :
     (string t ; int i ; trans indit string(f, t, format, p, fr) ;
      z := (char in string("|", i, t) |
           (L string real(t[1 : i - 1]) | L string real(t[i + 1 :: 1]))))
e) proc % trans indit string =
     (file f, ref string t, [1 : int m] char format,
      ref int p, [] int frame) :
     begin int n ; bool supp ; char k ; string x("") ;
           op ? = (string s) bool :
           (req insertion(format, p) ; p > m | false |
            int q = p ; replicator(format, p, n) ;
            (supp := format[p] = "s" | p plus 1) ;
            (char in string(format[p], loc int, s) | true |
            p := q ; false)) ;
           op ! = (char c) : (x plus(supp | c |: next = c | c)) ;
           proc next = char expr(get string(f, k) ; k) ;
           proc intreal mould = expr
           (? "r" ; sign mould(frame[3]) ; int mould ;
            (? "." | ! "." ; int mould) ;
            (? "e" | ! "10" ; sign mould(frame[6]) ; int mould)) ;
           proc sign mould = (int sign) : (sign > 0 |
            (? "z" | bool zs(true), sk(false) ; string t("") ;
             to n + 1 do(next = "." | sk := true ; (¬ zs | undefined) |
             zs := false ; t plus k) ;
             x plus(sign = 2 ∧ sk ∧ t[1] ≠ "-" | "+" + t | t) |
             x plus(sign = 2 ∧ next = "." | "+" | k)) ;
            p plus 1) ;
```



```
           proc int mould = expr
           (l : (? "z" | bool zs (true) ; to n do x plus
                (supp | "0" |: next = "." ∧ zs | "0" | zs := false ; k) ; l
                 ? "d" | to n do x plus(supp | "0" | next) ; l)) ;
           proc string mould = expr while ? "a" do to n do x plus
                (supp | "." | next) ;
      tis: (frame[1] = 6 | string mould |: intreal mould ; frame[1] > 3 |
            ! "|" ; intreal mould) ;
           t := x
     end ;
f) proc % trans indit choice =
     (file f, ref int c, string format, ref int p) :
     (req insertion(f, format, p) ; p plus 2 ; c := 1 ;
      while ¬ ask lit(f, format, p) do
      (c plus 1 ; format[p] = "," | p plus 1 | undefined) ;
      while format[p] ≠ ")" do(p plus 1 ; skip lit(format, p)) ;
     p plus 1 ; req insertion(f, format, p)) ;
g) proc % trans indit bool =
     (file f, ref bool b, string format, ref int p) :
     (req insertion(f, format, p) ; (format[p + 1] = "(" |
       p plus 2 ; (b := ask lit(f, format, p) |
         p plus 1 ; skip lit(format, p) |: ¬ ask lit(f, format, p) |
         undefined) |
       char k ; get string(f, k) ; b := (k = "1" | true |:
         k = "0" | false)) ;
     p plus 1 ; req insertion(f, format, p)) ;
h) proc % req insertion = (file f, [1 : int l] char s, ref int p) :
     while(p > l | false |: do align(f, s, p) | true |
       req lit(f, s, p)) do skip ;
i) proc % req lit = (file f, string s, ref int p) bool :
     (int q = p ; int n ; replicator(s, p, n) ;
          (s[p] = """" | int r = p ; to n do(p := r ;
          while(s[p plus 1] = """" | s[p plus 1] = """" | true) do
          (char k ; get string(f, k) ; k ≠ s[p] | undefined)) ; true |
                        p := q ; false)) ;
```

10.5.5. continued 3

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

10.5.6. Binary output

```
a) proc write bin = ([] outtype x) : put bin(stand back, x) ;
b) proc put bin = (file file, [1 : int n] outtype x) :
     if bin possible[chan of file] ∧ opened of file
     then if ¬ set possible[chan of file] thef state def of file
          then(state get of file ∨ ¬state bin of file | undefined)
          else state def of file := state bin of file := true ;
               state get of file := false
          fi ;
          for k to n do
          ([1 : int l] simplout y = straightout x[k] ;
          for j to l do
          ([1 : int m] int bin = to bin(file, y[j]) ;
          for i to m do(next plc(file) ;
          book of bfile of file[page of file, line of file,
               char of file] := bin[i])))
     else undefined
     fi ;
c) proc % to bin = (file f, simplout x) [] int :
     c a value of mode 'row of integral' whose lower bound is one,
       and whose upper bound depends on the value of 'f' and on the
       mode of the value of 'x'; furthermore,
       x = from bin(f, to bin(f, x)) c ;
d) proc % from bin = (file f, [] int y) simplout :
     c a value, if one exists, of a mode from which that specified by
       simplout is united, such that y = to bin(f, from bin(f, y)) c ;
```

10.5.7. Binary input

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

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

11. Examples

11.1. Complex square root

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

```
a) proc compsqrt = (compl z) compl : c the square root whose real part
       is nonnegative of the complex number z c
b) begin real x = re z, y = im z ;
c) real rp = sqrt((abs x + sqrt(x ↑ 2 + y ↑ 2))/2) ;
d) real ip = (rp = 0 | 0 | y/(2 × rp)) ;
e) (x ≥ 0 | (rp ⊥ ip) | (abs ip ⊥ (y ≥ 0 | rp | -rp)))
f) end compsqrt
```

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

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

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

11.3. Innerproduct2

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

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

Real-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 innerproduct3 = (ref int i, int n, proc real xi, yi) real :
       comment the innerproduct of two vectors whose n elements are the
       values of the expressions xi and yi and which depend, in general,
       on the value of i comment
b) begin long real s(long 0) ;
c) for k to n do(i := k ; s plus leng xi × leng yi) ;
d) short s
e) end 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-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 valu
       of the m by n matrix a is assigned to y, and the subscripts of this
       element to i and k comment
c) begin y := -1 ;
d) for p to m do for q to n do
e) if abs a[p, q] > y then y := abs a[(i := p), (k := q)] fi
f) end absmax
```

Statement-calls {8.7.1.c} using *absmax*:

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

11.6. Euler summation

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

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

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

11.8. Determinant of a matrix

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

An expression-call using *det*:

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

11.9. Greatest common divisor

An example of a recursive procedure:

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

An expression-call using *gcd*:

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

11.10. Continued fraction

An example of a recursive operation:

```
a) op / = ([1 : int n] real a, b) real :
     comment the value of a/b is that of the continued fraction
     a1/(b1 + a2/(b2 + ... an/bn)...) comment
b)   (n = 1 | a[1]/b[1] | a[1]/(b[1] + a[2 :: 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 = = (form a, ref const b) bool :
     (ref const ec ; (ec ::= a | val ec :=: b | false)) ;
j) op + = (form a, b) form :
     (a = zero | b |: b = zero | a | triple(a, plus, b));
k) op - = (form a, b) form : (b = zero | a | triple(a, minus, b));
l) op × = (form a, b) form :
     (a = zero ∨ b = zero | zero |: a = one | b |: b = one | a |
                                   triple(a, times, b));
m) op / = (form a, b) form :
     (a = zero ∧ ¬(b =: zero) | zero |: b = one | a | triple(a, by, b));
n) op ↑ = (form a, ref const b) form :
     (a = one ∨(b ↑=↑ zero) | one |: b :=: one | a | triple(a, to, b));
o) proc derivative of = (form e, c with respect to c ref var x) form :
p) begin ref const ec ; ref var ev ; ref triple et ; ref call ef ;
q) if   ec :: e then zero
r) elsf ev ::= e then(val ev :=: x | one | zero)
s) elsf et ::= e then
t)         form u = left operand of et, v = right operand of et,
u)         udash = derivative of (u, c with respect to c x),
v)         vdash = derivative of (v, c with respect to c x) ;
w)         case operator of et in
x)           udash + vdash, udash - vdash,
y)           u × vdash + udash × v, (udash - et × vdash)/v,
z)           v × u ↑ const(ec ::= v ; value of ec - 1) × udash
           esac
```

11.11. continued

```
aa) elsf ef ::= e then
ab)         ref function f = function name of ef;
ac)         form g = parameter of ef;
ad)         ref var y = bound var of f;
ae)         function fdash(y, derivative of(body of f, y));
af)         call(fdash, g) × derivative of(g, x)
ag) fi
ah) end derivative;

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

```
) begin   struct book = (string title, ref book next),
                auth = (string name, ref auth next, ref book book) ;
          ref book book ; ref auth auth, first auth(nil), last auth ;
          string name, title ; int i ; file input, output ;
          format format = fx30al,80alf ;
          proc update = expr if val first auth :=: nil
          then auth := first auth := last auth := auth(name, nil, nil)
          else auth := first auth ; while val auth :≠: nil do
                  (name = name of auth | known | auth := next of auth) ;
               last auth := next of last auth := auth :=
                auth(name, nil, nil) ;
     known: skip fi ;
       open(input, remote in) ; open(output, remote out) ;
       out(output, fp
          "to.enter.a.new.author,.type.""author"",.
           a.space,.and.his.name."l
          "to.enter.a.new.book,.type.""book"",.a.space,.
           the.name.of.the.author,.a.new.line.and.the.title."l
          "for.a.listing.of.the.books.by.an.author,.type.""list"",.
           a.space,.and.his.name."l
          "to.find.the.author.of.a.book,.type.""find"",.
           a.new.line.and.the.title."l
          "to.end,.type."5a"."lf, """end""") ;
client: in(input, fc("author","book","list","find","end","")f, i) ;
       case i in author, publ book, list, find, end, error esac ;
author: in(input, format, name) ; update ; client ;
 publ
 book: in(input, format, (name, title)) ; update ;
       if val book of auth :=: nil then book of auth :=: book(title, nil
       else book := book of auth ; while val next of book :≠: nil do
            (title = title of book | client | book := next of book) ;
            (title ≠ title of book | next of book := book(title, nil))
       fi ; client ;
```



```
aa)   list: in(input, format, name) ; update ;
ab)         out(output, fp"author:."30allf, name) ;
ac)         if val first of auth :=: nil
ad)         then put(output, "no.publications")
ae)         else while val book :≠: nil do
af)              begin if line number(output) = max line[remote out]
ag)                   then out(output, f41k"continued.on.next.page"p
                                "author:."30a41k"continued"llf, name)
ah)                   fi ; out(output, f80alf, title of book) ;
ai)                   book := next of book
aj)              end
ak)         fi ; client ;
al)   find: in(input, fl80alf, title) ; auth := first auth ;
am)         while val auth :≠: nil do
an)              begin book := book of auth ; while val book :≠: nil do
ao)                   if title = title of book
ap)                   then out(output, fl"author:."30af,
                                name of auth) ; client
aq)                   else book := next of book
ar)                   fi ; auth := next of auth
as)              end ; to 2 do new line(output) ;
at)         put(output, "unknown") ; client ;
au)    end: new page(output) ; put(output, "signed.off") ;
av)         close(input) ; close(output).
aw)  error: new line(output) ; put(output, "mistake,.try.again.") ;
ax)         new line(input) ; client
ay)   end   authors and titles enquiry system
```

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

12. Glossary

Given below are the locations of the first, and sometimes other, instructive appearances of a number of words which, in Chapters 1 up to 10 of this Report, have a specific technical meaning. A word appearing in different grammatical forms (e.g. "conversion", "convert", "converted", "converting") is given once, usually as infinitive (e.g. "convert").

{Denn eben, wo Begriffe fehlen,
Da stellt ein Wort zur rechten Zeit sich ein.
Faust, J.W. von Goethe.}

EE. Ephemeral Epilogue

EE.1. Errata

{Cuiusvis hominis est errare, nullius,
nisi insipientis in errore perseverare.
Orationes Phillippicae, M.T. Cicero.}

EE.1.1. Syntax

a) errata : erratum sequence.
b) erratum : location list, change.
c) location : line number ; fragment.
d) line number : set off, shift option.
e) set off : section, paragraph option.
f) section : integral denotation, point symbol, section option.
g) paragraph : TAG.
h) shift : plusminus, integral denotation.
i) fragment : top, up to symbol, bottom.
j) top : line number.
k) bottom : line number.
l) change : instead of symbol, old text denotation,
substitute symbol, new text, please symbol.
m) old text denotation : old text ; begin of old text,
query symbol, query symbol, query symbol, end of old text.

EE.1.2. Representations

symbol	representations
instead of symbol	*
substitute symbol	→
please symbol	*
query symbol	?

{Examples:
b) 15.2.3.c-7 * x.y → x.z * ;
14.2., 14.2.+3, 14.2.+7, 15.2.3.c-7 * a → b * ;
0.:11.11.bc * Introduction ??? example. → * ;
c) 15.2.3.c-7 ; 0.:11.11.bc ;
d) 14.2. ; 14.2.+3 ;
e) 14.2. ; 15.2.3.c ;
f) 14.2. ;
g) c ;

EE1.2. continued

h) +3 ; -7 ;

i) 0.:11.11.bc ;

j) * x.y → x.z * ;

k) x.y ; Introduction ??? example. }

EE. 1.3. Semantics

a) Errata are elaborated in the following steps:

Step 1: The errata are considered ;

Step 2: If the considered errata contain a location-list containing a comma-symbol, then other errata are considered instead which are the same sequence of Report-tokens as would be obtained by replacing that comma-symbol in the considered errata by the change following that location-list, and Step 2 is taken ;

Step 3: The constituent erratums of the considered errata are elaborated collaterally.

An erratum whose constituent location-list does not contain a comma-mbol is elaborated in the following steps:

ep 1: The erratum is considered ;

ep 2: That erratum is considered instead which is the same sequence of Report-tokens as would be obtained by replacing each empty constituent shift-option in the considered erratum by a plus-symbol followed by a ero-symbol ;

p 3: That erratum is considered instead which is the same sequence of eport-tokens as would be obtained by replacing, in the considered rratum, each location not containing an up-to-symbol by {the line-umber which is} that location followed by an up-to-symbol followed by he line-number which is} that location ;

4: Letting lt(lb) stand for the line of the Report possessed by the t-off contained in the top (bottom) contained in the considered ratum, vt(vb) for the value of the integral-denotation and st(sb) for e plusminus contained in the shift contained in the top (bottom) ntained in the considered erratum, then an upper line (lower line) sought in the Report at a distance of vt(vb) lines from lt(lb) in direction of the end or begin of the Report depending upon whether sb) is a plus-symbol or minus-symbol, and, if the upper line and er line are found and if the upper line is not closer to the end

EE. 1.3 continued

of the Report than the lower line, then the set of consecutive lines the first (last) of which is the upper line (lower line) is considered, and Step 5 is taken; otherwise, the further elaboration is undefined. {and information from outside this Report, eg the reader's intelligence, must be taken into account.}

Step 5: If the constituent old-text-denotation of the constituent change of the considered erratum is an old-text and the considered set of lines comprises exactly one sequence of Report-tokens which is the same as that old-text, or if that old-text-denotation is not an old-text and the considered set of lines comprises exactly one sequence of Report-tokens which is the constituent begin-of-old-text of that old-text-denotation followed by a nonempty sequence of Report-tokens followed by the constituent end-of-old-text, then that sequence of Report-tokens is replaced by a new appearance of the sequence of Report-tokens which is the constituent new-text of the constituent change of the considered erratum, and Step 6 is taken; otherwise, the further elaboration is undefined ; {and information from outside this Report, eg the reader's intelligence must be taken into account.}

Step 6: The title of the Report and all titles of the Chapter, Section and paragraph comprising the considered lines are made to refer to the Report, Chapter, Section and paragraph as modified in Step 5, and the elaboration of the erratum is complete.

EE.2. Correspondence with the Editor

EE.2.1. Example of a letter to the Editor

Amsterdam, 16 Feb. 1968

{Achève, petit Jean; c'est fort bien débuté.
Les Plaideurs, J. Racine.

Dear Editor,

This morning I received with thanks the Draft Report on ALGOL 68 and read it with interest. Please convey my feelings of deep appreciation to your co-authors for the fine work that they achieved. I take the liberty to suggest the following amendments, which are of a purely descriptional nature, for the final Report:

EE.1.4. Extensions.

a) If the constituent set-offs of two line numbers separated by a comma-symbol or up-to-symbol are the same, then the second set-off may be omitted.

2.1. continued

.1.u, 5.4.1.b, 7.1.1.z, 8.1.1.b, 8.1.1.i, 8.2.1.1.d, 8.2.2.1.a, 8.7.1.b+1
* delivering a → *
.2. continued +2:3, 8.2.0.1. continued +12, 8.2.0.1. continued +24,
.0.1. continued +25, 8.2.0.1. continued +26, 8.2.0.1. continued +27,
.2.+8, 11.1.+2, 11.2.+2, 11.2.+3, 11.2.+4, 11.3.+3, 11.4.+4
* delivering-a- → *
4.3.+3 * delivering-a-???-delivering-a →
real-parameter-and-a-procedure *
1.q, 1.2.1.u, 1.2.2.h, 1.2.4.n, 1.2.4.n+1, 4.3.1.b, 8.1.1.a, 8.1.1.c,
1.h+1, 8.1.1.j * DELIVETY → MODETY *
1.h * DELIVETY : dep DELIVETY → MODETY : dep MODETY *
2.k * k)???. → *
1.1.b * called???. →
called MODETY FORM : fitted procedure MODETY FORM. *
1.1.c * c)???. → *

These amendments do not change the language at all, but do diminish
number of rules by two and, moreover, delete over three hundred
ingless letters. Of course, the deletion of rules 1.2.2.k and
1.1.c will cause some changes in the paragraphs of the rules
owing them, in the examples and, possibly, in some cross references.
d you be so kind to take care of this yourself if you accept these
dments?

f you answer this letter at all, would you then tell me why you
not unify statements and expressions by considering statements as
ty-expressions like in Orthogonal Design [4]?

Yours sincerely,

A. van W.

EE.2.2. Reply by the Editor to the letter in EE.2.1

Amsterdam, 16 Feb. 1968.

Dear A. van W.,

Many thanks for your prompt reaction to the receipt of the Draft Report. As to your amendments, I fully agree with them; actually, I had also found these very changes myself, but too late for incorporation in the text. I shall do my best to have them incorporated in the final Report and, of course, to make the necessary changes you asked for in paragraphs, examples and cross references.

Now about the statements. We had much trouble understanding the coercion process, dear me; especially skip, jump and nihil caused us a lot of ambiguous parsings. At an early stage, we separated the statements from the expressions again because we thought that this alleviated the problem. Now that the tempest has passed, I see that you can very well unify them again to great advantage. I shall sketch it for you, since you seem so much interested in our work. The errata are assumed to be elaborated after yours:

6.0.1.h * . → .
i) SOME statement : SOME void expression. *
1.2.1.u * u)???. → u) RESULT : MODE ; void. *
1.2.1.q , 1.2.4.a, 1.2.4.n, 1.2.4.n+1 * MODETY → RESULT *
1.2.3.b * b)???. → b) CLAUSE : RESULT expression. *
6.0.1.e * SOME statement. → SOME void expression. *
6.1.1.e+1 * statements → void expressions *
6.2:6.3.-1 * Unitary statements???6.3.2.a).} → *
7.1.1.w, 7.1.1.x, 7.1.1.z+1 * tail → void tail *
8.0.1.a:8.0.1.e+1 * a)???relation. →
a) COERCETY unitary RESULT expression:
COERCETY RESULT formary ; COERCETY RESULT confrontation.
b) COERCETY RESULT formary :
COERCETY RESULT ADIC formulation ; COERCETY RESULT primary.
c) COERCETY MODE ADIC formulation : COERCETY MODE ADIC formula.
d) void ADIC formulation : void ADIC formula ; called void ADIC formula ;
NONPROC ADIC formula ; called NONPROC ADIC formula.
e) hip FORCED MODE ADIC formula : FORCED MODE ADIC formula.

EE.2.2. continued

f) void confrontation : MODE confrontation.
g) hip FORCED MODE confrontation : FORCED MODE confrontation.
h) MODE confrontation : MODE assignation ;
MODE conformity relation ; MODE identity relation. *
8.1.1.a:8.1.1.j * a)???dep MODETY. →
a)* COERCETY formula : COERCETY RESULT ADIC formula.
b) RESULT PRIORITY formula : LMODE PRIORITY operand,
procedure with a LMODE parameter and a RMODE parameter
RESULT PRIORITY operator, RMODE PRIORITY plus one operand.
c)* operand : MODE ADIC operand.
d) MODE PRIORITY operand :
adjusted MODE PRIORITY formula ; MODE PRIORITY plus one operand.
e) MODE PRIORITY NINE plus one operand : MODE monadic operand.
f) MODE monadic operand :
adjusted MODE monadic formula ; hip adjusted MODE primary.
g) RESULT monadic formula : RESULT dep ; procedure with a RMODE
parameter RESULT monadic operator, RMODE monadic operand.
h) MODE dep :
value of symbol, peeled reference to MODE monadic formula ;
value of symbol, hip peeled reference to mode primary.
i)* depression : MODE dep. *
8.2.1.1.b:8.2.1.1.d * b)???d) →
b) called COERCEND : fitted procedure COERCEND.
c) *
8.2.1.1.e * e) → d) *
8.2.2.1.b * b)???. → *
8.3.1.a:8.3.1.a+1 * a)???cohesion. →
a) COERCETY RESULT primary : COERCETY CLOSED RESULT expression ;
COERCETY RESULT cohesion ; RESULT call. *
8.3.1.e * e)???. →
e) skip : skip symbol.
f) jump : goto symbol option, label identifier.
g) reference to MODE nihil : nil symbol.
h) void cohesion : NONPROC cohesion ; void expression call ; skip ; jump.
i) void call : called void cohesion ; called NONPROC cohesion. *

EE.2.2. continued 2

Of course, some examples, cross references and section headings must be modified appropriately, and the scanty Semantics of 6.2.2 must be inserted into 8.3.2, where it fits much better.
The effect is quite satisfying: one rule less, a whole section {6.2} gone, and a much cleaner set up. This is, of course, also for the final Report.

Yours for ever,
Editor.

EE.2.3. Second example of a letter to the Editor

Somewhere, 1 June 1968.

Sir,
it stinks,
yours,
A.N. Onymous.

EE.2.4. Reply option by the Editor to the letter in EE.2.3

{Empty(the letter was not received in time, see PP.3.b). }

EE.2.5. Third example of a letter to the Editor

From Amsterdam to Calgary, 30 Jan. 1968.

Dear Editor,
Please consider the following errata to have been elaborated:

1.2.4.h+3 * Rule c → Rule d *
2.2.+3 * Carroll → Carroll. *
2.2.+4 * "computer" → computer *
2.2.2.1+2 * {8.8.2.c} → (8.8.2.c) *
2.2.3.1.b+4, 2.2.3.5.a * called → *
2.2.3.1.c+6 * divided → "divided *
2.2.5.+3 * either → *
3.1.1.c+6 * ≠ → ≠ *
3.1.1.d-1 * :≠: → :≠: *
3.1.2.d+2 * cpaital → capital *
4.1.1.e+3 * 1.2.2.t → 1.2.2.s *
4.1.2.a+6 * "is → is" *
4.4.+4 * "ALGOL 68 programs" → "ALGOL-68" programs *

2.5. continued

.3.e-5 * *max* → *max* = *

.Step 5+3 * each → the first and last constituent *

.2.a+3 * successor → "successor" *

.2.c * 10.4.b → 10.4.a *

.2.c+2 * 10.4.a → 10.4.b *

.1.c+1 * option → *

.1.p+1 * . → ; EMPTY. *

.2.b+1:3 * either???symbols → an actual-declarer or formal-declarer *

.2.-1 * 9.2b → 9.2.b *

.1.d * ADIC formula → confrontation *

.2. continued+1 * operator-tokens as → operators *

.0.1. continued+9 * 8.1.1.b, → 8.1.1.b, c *

.0.1. continued+14, 8.2.0.1. continued+15, 8.2.0.1. continued+16,

.0.1. continued+17 * 8.1.1.c → 8.1.1.e *

.0.1. continued+18 * 8.1.1.d → 8.1.1.f *

0.1. continued+19 * 8.1.1.e → 8.1.1.g *

0.1. continued+26 * 8.3.1.c → 8.3.1.b *

2.2. continued+13 * coercend → conditional-expression *

2.2. continued+13 * *(x)* → the routine *(x)* or the routine *

2.2. continued+16 * coercend → serial-expression *

6.2.-1 * *esac*) → *esac*)} *

2.d+7 * structure (8.5.1.f) → veld (8.5.1.g) *

1.f * : → : MODE veld.

g) MODE veld : *

2.-1 * f) → g) *

2.a+1 * expressions and structures → velds *

2.a+2 * ; → , where the elaboration of a structure-pack is that of its constituent structure ; *

2.b+2:3 * , the???; → ; *

26 * empty. → empty.

d) Each representation of a symbol appearing in Sections 9.1 up to 9.5 may be replaced by any other representation, if any, of the same symbol. *

b.-1 * *skip*) → *skip*)) *

1.+8 * mode → made *

2.b+14, 10.5.3.c-4 * *: int n → 1 : int n* *

EE.2.5. continued 2

10.5.2.b+16 * *1 : int n → : int n* *

10.5.3.b+18 * *string → string* *

10.5.3.c-2 * *get???≠ → ¬?* *

11.11. * manipluation → manipulation *

. We plan to submit, if necessary, (a) loose-leaf letter(s) comprising other errata.

Collegialiter,

The authors.

EE.2.6. Reply by the Editor to the letter in EE.2.5

Amsterdam, 31 Jan. 1968.

Dear Authors,

The errata comprised in your letter of 30 Jan. 1968 and also those, possibly, comprised in (an) other letter(s) by you if communicated to the readers of the ALGOL Bulletin, are considered to have been elaborate

Ours,

Editor.