ADAPTED : ADJUSTED ; widened ; rowed ; 
hipped ; voided.
ADIC : PRIORITY ; monadic.
ADJUSTED : FITTED ; procedure ; united.
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.
ANY : KIND ; suppressible KIND ;
replicable KIND ;
replicable suppressible KIND.
BITES : structured with row of boolean 
field LENGTHENETY letter aleph.
BOX : LIMODESETY box.
BYTES : structured with row of character 
field LENGTHENETY letter aleph.
CLAUSE : MODI clause.
CLOSED : closed ; collateral ; conditional.
COERCED : MODI FORM.
COMPLEX : structured with real field 
letter r letter e and real field 
letter l letter m.
DIGIT : digit FIGURE.
EIGHT : SEVEN plus one.
EMPTY :.
FEAT : firm ; weak ; soft.
FIELDS : MODI field TAG.
FIELD : MODI field TAG.
FIGURE : zero ; one ; two ; three ; four ; 
five ; six ; seven ; eight ; nine.
FITTED : dereferenced ; deprocedured.
FIVE : FOUR plus one.
FORESE : ADIC formula ; cohesion ; base.
FORM : confrontation ; FORESE.
FOUR : THREE plus one.
INTEGER : LONGESTY integral.
INTERNAL : INTEGER ; REAL.
KIND : sign ; zero ; digit ; point ;
exponent ; complex ; string ; character.
LENGTH : letter l letter o letter n 
letter g.
LENGTHENETY : LENGTH LENGTHENETY ; EMPTY.
LETTER : letter ALPHA ; letter aleph.
LEFTFIELDSETY : FIELDS and ; EMPTY.
LIST : list ; sequence.
MODE : MODE.
MOD : MOD and.
MODS : INMOD ; IMODS IMOD.
MODINSETY : MODE and IMODSETY ; EMPTY.
MODM : MOD and.
LONGESTY : long LONGESTY ; EMPTY.
LOGISTIC : IMODSETY.
LOWER : lower ; upper.
LABEL : MODE mode ; label.
MODE : MODE ; UNITED.
MOID : MODE ; void.
MOOD : TYPE ; STOWED.
NINE : EIGHT plus one.
NONPROC : PLAIN ; format ; procedure 
with PARAMETERS MOD ; reference to 
NONPROC ; structured with FIELDS ; 
row of NONPROC ; UNITED.
NOW : nonSTOWED ; structured with 
FIELDS.
NOWSTOWED : TYPE ; UNITED.
NOTION : ALPHA ; NOTION ALPHA.
NUMBER : one ; two ; three ; four ; 
five ; six ; seven ; eight ; nine.
PACK : pack ; package.
PARAMETER : MODE parameter.
PARAMETERS : PARAMETER ;
PARAMETERS and PARAMETER.
PARAMETER : with PARAMETERS ; EMPTY.
PHRASE : declaration ; CLAUSE.
PLAIN : IMODE ; boolean ; character.
PMAN : procedure with IINMODE parameter 
and IPAND parameter MOD ; 
procedure with INMODE parameter MOD.
PRIMITIVE : integral ; real ; boolean ; 
character ; format.
PRIORITY : priority NUMBER.
PROCEDURE : procedure PARAMETER MOD.
REAL : LONGESTY real.
REPEIT : reference to ; EMPTY.
RIGHTFIELDSETY : and FIELDS ; EMPTY.
RING : MODE.
RINGOSES : PROCEDURE and MODE ; EMPTY.
ROWS : row of ; ROWS row of.
ROWSITY : ROWS ; EMPTY.
ROWWITY : ROWSITY.
SEPARATOR : LIST separator ; 
go on symbol ; completer ; sequencer.
SIX : SIX plus one.
SOME : serial ; unitary ; CLOSED ;
choice ; THEELSE.
SORT : strong ; FEAT.
SORTITY : SORT ; EMPTY.
SORTM : strong ; firm.
STOWED : structured with FIELDS ; 
row of MODE.
STRING : row of character ; character.
STRENGTH : strong ; EMPTY.
TAG : LETTER ; TAG LETTER ; TAG DIGIT.
THEELSE : then ; else.
THREE : TWO plus one.
TWO : one plus one.
TYPE : PLAIN ; format ; PROCEDURE ; 
reference to MODE.
UNITED : union of IMODS MODE mode.
VICTAL : VIBRAC ; format.
VIBRAC : virtual ; actual.
This Report has been reviewed by Technical Committee 2 on Programming Languages and approved for publication by the General Assembly of the International Federation for Information Processing. Reproduction of the Report, for any purpose, but only of the whole text together with the cover-letter, is explicitly permitted without formality.

Report of IFIP TC 2 to the General Assembly of IFIP
in the matter of the Algorithmic Language ALGOL 68

Attached will be found the first Report on the Algorithmic Language ALGOL 68, submitted with a recommendation for publication under IFIP auspices by Technical Committee 2 in response to a request from Working Group 2.1 (ALGOL).

This Working Group has been directed, by TC 2, for a number of years towards concern for the design of common programming languages. During this time, TC 2 has realized the magnitude and difficulty of this task.

In the course of pursuing its responsibilities, WG 2.1 commissioned and has guided the work of the four listed authors of the attached Report and has stated its desire to acknowledge the considerable effort devoted by these men to their task. This Report, however, must be regarded as constituting more than simply the work of four authors. Its content has been influenced throughout and the results are, in the main, a consequence of discussions within the Working Group.

The Report is, thus, submitted by TC 2 as representing the current consolidated outcome of WG 2.1 activity. This does not imply that any member of the Working Group necessarily agrees with every aspect of the undertaking nor guarantees that all relevant aspects of the problem have been considered. Indeed, fear has been expressed, in the form of a Minority Report from the Working Group to TC 2, that the direction taken in the attached Report will not lead towards the goal of providing appropriate programming tools for the future. It has, however, been decided by the Working Group, that this work has reached the proper stage for submission to the crucial tests of implementation and subsequent use by the computing community. In this opinion, TC 2 concurs. Therefore, this Report is submitted for publication as representing one of the possible approaches to the subject, rather than in the spirit of a final answer.

WG 2.1 will be directed to keep continually under review experience obtained as a consequence of this publication, so that it may institute such corrections and revisions to the Report as become desirable. To this end, TC 2 adds its endorsement to the request of WG 2.1 that all who wish to contribute to this work should do so through the established medium of the ALGOL-Bulletin.

In consonance with these declared aims, TC 2 requests that the IFIP General Assembly take those actions necessary and proper to insure expeditions and widespread publication of this Report on the Algorithmic Language ALGOL 68 together with this page as a covering statement.
Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam, The Netherlands.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications; it 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.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.
REPORT ON THE ALGORITHMIC LANGUAGE ALGOL 68

A. VAN WIJNGAARDEN (EDITOR), B.J. MAILOUX, J.E.L. PECK and C.H.A. KOSTER

SECOND PRINTING BY THE MATHEMATISCH CENTRUM, AMSTERDAM, M 101, OCT. 1969

BIBLIOTHEEK MATHEMATISCH CENTRUM AMSTERDAM
Acknowledgements

(Habent sua fata libelli.
De litteris, Terentianus Maurus.)

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 in St Pierre de Chartreuse near Grenoble in October 1965, three reports describing more or less complete languages were amongst the contributions, by Niklaus Wirth [8], Gerhard Seegmüller [6], and Aad van Wijngaarden [9]. In [6] and [8], the descriptive technique of [3] was used, whereas [9] featured a new technique for language design and definition. Other significant contributions available were papers by Tony Hoare [2] and Peter Naur [4,5].

At meetings in Kootwijk near Amsterdam in April 1966, Warsaw in October 1966, Zandvoort near Amsterdam in May 1967, Tirrenia near Pisa in June 1968, North Berwick near Edinburgh in July 1968 and Munich in December 1968, 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 [10], later reinforced by John Peck [11,12], and finally by Kees Koster [13,14,15,16]. Versions were used during courses on the language held in Amsterdam [12], Bakuriani [13], Grenoble [13], Copenhagen [14], Esztergom [14], Calgary [14] and Chicago [16]. These courses served as test cases and the experience gained in explaining the language to skilled audiences and the reactions of the students influenced the succeeding versions.

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. At the risk of embarrassing omissions, special mention should be made of Jan Garwick, Jack Merner, Peter Ingerman and Manfred Paul for [1], the Brussels group consisting of M. Sintoff, P. Branquart, J. Levi and P. Wodon for numerous brainstormings, A.J.M. van Gils of Apeldoorn, G. Goos and his group of Munich, also for [7], G.S. Tseitin of Leningrad and L.G.L.T. Meertens and J.W. de Bakker of Amsterdam. An occasional choice of a, not inherently meaningful, identifier in the sequel may compensate for not mentioning more names in this section.

Acknowledgements continued

Contents

0. Introduction
  0.1. Aims and principles of design
    0.1.1. Completeness and clarity of description. 0.1.2. Orthogonal design.
    0.1.3. Security. 0.1.4. Efficiency. 0.1.4.1. Static mode checking.
    0.1.4.2. Static scope checking. 0.1.4.3. Mode independent parsing.
    0.1.4.4. Independent compilation. 0.1.4.5. Loop optimization.
    0.1.4.6. Representations
  0.2. Comparison with ALGOL 60
    0.2.1. Values in ALGOL 68. 0.2.2. Declarations in ALGOL 68. 0.2.3. Dynamic storage allocation in ALGOL 68. 0.2.4. Collateral elaboration in ALGOL 68. 0.2.5. Standard declarations in ALGOL 68. 0.2.6. Some particular constructions in ALGOL 68

1. Language and metalanguage
  1.1. The method of description
    1.1.1. The strict, extended and representation languages. 1.1.2. The syntax of the strict language. 1.1.3. The syntax of the metalanguage. 1.1.4. The production rules of the metalanguage. 1.1.5. The production rules of the strict language. 1.1.6. The semantics of the strict language. 1.1.7. The extended language. 1.1.8. The representation language
  1.2. The metaproduction rules
    1.2.1. Metaproduction rules of modes. 1.2.2. Metaproduction rules associated with modes. 1.2.3. Metaproduction rules associated with phrases and coercion. 1.2.4. Metaproduction rules associated with coercions. 1.2.5. Other metaproduction rules
  1.3. Pragmatics

2. The computer and the program
  2.1. Syntax
  2.2. Terminology
    2.2.1. Objects. 0.2.2. Relationships. 2.2.3. Values. 2.2.3.1. Plain values. 2.2.3.2. Structured values. 2.2.3.3. Multiple values.
    2.2.3.4. Routines and formulas. 2.2.3.5. Names. 2.2.4. Modes and scopes.
    2.2.4.1. Modes. 2.2.4.2. Scopes. 2.2.5. Actions
  2.3. Semantics

3. Basic tokens and general constructions
  3.0. Syntax
    3.0.1. Introduction. 3.0.2. Letter tokens. 3.0.3. Denotation tokens.
    3.0.4. Action tokens. 3.0.5. Declaration tokens. 3.0.6. Syntactic tokens. 3.0.7. Sequencing tokens. 3.0.8. Hip tokens. 3.0.9. Extra tokens and comments. 3.0.10. Special tokens
  3.1. Symbols
    3.1.1. Representations. 3.1.2. Remarks
4. Identification and the context conditions
   4.1. Identifiers
      4.1.1. Syntax. 4.1.2. Identification of identifiers
   4.2. Indications
      4.2.1. Syntax. 4.2.2. Identification of indications
   4.3. Operators
      4.3.1. Syntax. 4.3.2. Identification of operators
   4.4. Context conditions
      4.4.1. The identification conditions. 4.4.2. The uniqueness conditions.
      4.4.3. The mode conditions. 4.4.4. The declaration condition

5. Denotations
   5.1. Plain denotations
      5.1.1. Integral denotations. 5.1.2. Real denotations. 5.1.3. Boolean
             denotations. 5.1.4. Character denotations
   5.2. Bits denotations
   5.3. String denotations
   5.4. Routine denotations
   5.5. Format denotations
      5.5.1. Syntax. 5.5.2. Syntax of integral patterns. 5.5.3. Syntax of real
             patterns. 5.5.4. Syntax of boolean patterns. 5.5.5. Syntax of character
             patterns. 5.5.6. Syntax of complex patterns. 5.5.7. Syntax of string
             patterns. 5.5.8. Transforms

6. Phrases
   6.1. Serial clauses
   6.2. Collateral phrases
   6.3. Closed clauses
   6.4. Conditional clauses

7. Unitary declarations
   7.1. Declarers
   7.2. Mode declarations
   7.3. Priority declarations
   7.4. Identity declarations
   7.5. Operation declarations

8. Unitary clauses
   8.1.1. Syntax
   8.2. Coercends
      8.2.1. Dereferenced coercends. 8.2.2. Deaprocedured coercends.
      8.2.3. Proceeded coercends. 8.2.4. United coercends. 8.2.5. Widened
             coercends. 8.2.6. Rowed coercends. 8.2.7. Hipped coercends.
      8.2.8. Voided coercends
   8.3. Confrontations
      8.3.1. Assignations. 8.3.2. Conformity relations. 8.3.3. Identity rela-
             tions. 8.3.4. Casts
   8.4. Formulas
   8.5. Cohesions
      8.5.1. Generators. 8.5.2. Selections
   8.6. Bases
      8.6.1. Slices. 8.6.2. Calls
Contents continued 2

9. Extensions
   9.1. Comments
   9.2. Contractions
   9.3. Repetitive statements
   9.4. Contracted conditional clauses

10. Standard prelude and postlude
   10.1. Environment enquiries
   10.2. Standard priorities and operations
         10.2.0. Standard priorities. 10.2.1. Rows and associated operations.
         10.2.2. Operations on boolean operands. 10.2.3. Operations on integral
         operands. 10.2.4. Operations on real operands. 10.2.5. Operations on
         arithmetic operands. 10.2.6. Operations on character operands.
         10.2.7. Complex structures and associated operations. 10.2.8. Bit struc-
         tures and associated operations. 10.2.9. Bytes and associated opera-
         tions. 10.2.10. Strings and associated operations. 10.2.11. Operations
         combined with assignments
   10.3. Standard mathematical constants and functions
   10.4. Synchronization operations
   10.5. Transput declarations
         10.5.0. Transput modes and straightening. 10.5.0.1. Transput modes.
         10.5.0.2. Straightening. 10.5.1. Channels and files. 10.5.1.1. Channels.
         10.5.1.2. Files. 10.5.1.3. Standard channels and files. 10.5.2. Format-
         less transput. 10.5.2.1. Formatless output. 10.5.2.2. Formatless input.
         10.5.3. Formatted transput. 10.5.3.1. Formatted output. 10.5.3.2. For-
         matted input. 10.5.4. Binary transput. 10.5.4.1. Binary output.
         10.5.4.2. Binary input
   10.6. Standard postlude

11. Examples
   11.1. Complex square root
   11.2. Innerproduct 1
   11.3. Innerproduct 2
   11.4. Innerproduct 3
   11.5. Largest element
   11.6. Euler summation
   11.7. The norm of a vector
   11.8. Determinant of a matrix
   11.9. Greatest common divisor
   11.10. Continued fraction
   11.11. Formula manipulation
   11.12. Information retrieval
   11.13. Cooperating sequential processes
   11.14. Towers of Hanoi

12. Glossary
   12.1. Technical terms
   12.2. Paranojons
0. Introduction

0.1. Aims and principles of design

a) In designing the Algorithmic Language ALGOL 68, Working Group 2.1 on ALGOL of the International Federation for Information Processing expresses its belief in the value of a common programming language serving many people in many countries.

b) ALGOL 68 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, accepted the following as their aims:

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. The method adopted in this Report is based upon a strict language comprising a language core, whose description is minimized. The remainder of the language is described in terms of this core, thereby reducing the amount of semantic description at the cost of a heavier burden on the syntax. It is recognized, however, that this method may be difficult for the uninitiated reader. Therefore, a companion volume, entitled "Informal Introduction to ALGOL 68", has been prepared at the request of the Group by C.H. Lindsey and S.G. van der Meulen, and further companion volumes on specific aspects of the language may well follow.

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, the use of which is required only in those cases in which the programmer explicitly makes use of the flexibility offered by the unified mode feature.

0.1.4.2. Static scope checking

The syntax of ALGOL 68 is such that no scope checking during run time is necessary except in some cases in which the programmer explicitly makes use of the flexibility offered by the absence of syntactical scope restrictions.

0.1.4.3. Mode-independent parsing

The syntax of ALGOL 68 is such that the parsing of a program can be performed independently of the modes of its constituents. Moreover, there is an algorithm which determines in a finite number of steps whether an arbitrary given sequence of symbols is a proper program.

0.1.4.4. Independent compilation

The syntax of ALGOL 68 is 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.

0.1.4.5. 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.

0.1.4.6. Representations

Representations of ALGOL 68 symbols have been chosen so that the language may be implemented on computers with a minimal character set. At the same time implementers may take advantage of a larger character set, if it is available.

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 com-
pletely new features. It is, therefore, difficult to isolate differences be-
tween 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 and Boolean,
ALGOL 68 features an infinity of "modes", i.e., generalizations of the con-
cept "type".

b) Each plain value is either arithmetic, i.e., of integral or real mode
and then it is of one of several lengths, or it is of boolean 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 the character array, which corresponds ap-
proximately to the ALGOL 60 string; examples of structured values are com-
plex numbers, machine words considered as sequences of bits or of bytes, 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 procedure body is generalized in ALGOL 68 to the
concept of "routine", which includes also 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 com-
piled 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 sub-
ject 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 dec-
larations and procedure declarations, ALGOL 68 features the "identity decl-
laration" whose expressive power includes all of these, and more. In fact,
the identity declaration declares not only variables, but also constants, of
any mode, and, moreover, forms the basis of a highly efficient and powerful
parameter mechanism.

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

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

0.2.3. Dynamic storage allocation in ALGOL 68

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

0.2.4. Collateral elaboration in ALGOL 68

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

0.2.5. Standard declarations in ALGOL 68

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

0.2.6. Some particular constructions in ALGOL 68

a) The ALGOL 60 concepts of block, compound statement and parenthesized expressions 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 possess 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 or structured values from other, simpler values.

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

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 fail to find it wise. Grooks, Piet Hein.}
1. Language and metalanguage

1.1. The method of description

1.1.1. The strict, extended and representation languages

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

b) In the definition, the "English language" and a "formal language" are used. For both of these languages, as also for the strict language and the extended language, typographical and syntactic marks are used which bear no immediate relation to those used for the representation language.

1.1.2. The syntax of the strict language

a) The strict language is defined by means of a "syntax" and "semantics". This syntax is a set of "production rules" for "notions"; it is expressed in "small syntactic marks", in this Report "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" and "z";


"other syntactic marks", in this Report ",", ("point"), "," ("comma"), ":" ("colon"), ";" ("semicolon") and "*" ("asterisk").

(Note that these marks are in another type font than the marks in this sentence.)

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

c) A production rule for a given notion consists of that notion, possibly preceded by an asterisk, followed by a colon, a "list of notions" (see d), and a point, in that order. The list of notions is said to be a "direct production" of the given notion.

d) A list of notions is a nonempty sequence of "members" separated by commas; a member is either a notion and is then said to be "productive" {, or nonterminal,} or is a symbol {, which is terminal,} or is empty, or is some other protonotion (and then the production rule of whose list of notions it is a member is said to be a "blind alley").

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

f) A "terminal production" of a notion is a production of that notion each of whose members is either a symbol or empty.
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, the list of notions
'integral part option, fractional part'
is a direct production of the notion 'variable point numeral', and contains
two members, both of which are productive. A terminal production of this
same notion is
'digit zero symbol, point symbol, digit one symbol'.
The member 'digit zero symbol' is an example of a symbol, and is terminal.
The protonotion 'twas brillig and the slithy toves' is neither a symbol nor
a notion in the sense of this Report, in that it does not end with 'symbol'
and no production rule for it is given (1.1.5.b,c).]

1.1.3. The syntax of the metalanguage

a) Some production rules of the strict language are given explicitly and
others are obtained with the aid of a "metalanguage" whose syntax is a set
of production rules for "metanotions".

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

c) A production rule for a given metanotation consists of that metanotion
followed by a colon, a "list of metanotions" (see d), and a point, in that
order. The list of metanotions is said to be a direct production of the
given metanotation.

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

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

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

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

1.1.4. The production rules of the metalanguage

The production rules of the metalanguage are the rules obtained from the
rules in Section 1.2 in the following steps:
Step 1: If some rule contains one or more semicolons, then it is replaced by
two new rules, the first 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 follo-
ing its first semicolon, whereupon Step 1 is taken again;
1.1.4. continued

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

(For instance, the rule
'TAG : LETTER ; TAG LETTER ; TAG DIGIT.',
from 1.2.1.r, 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 might find it helpful to read ";" as "may be a", "," as "followed by a", and ";" as "or a".)

1.1.5. The production rules of the strict language

a) A production rule of the strict language is any rule obtained in the following steps from the rules given in Chapters 2 up to 8 inclusive in the sections whose heading is, or begins with, "Syntax":
Step 1: Identical with Step 1 of 1.1.4;
Step 2: One of the rules obtained is considered;
Step 3: If the considered rule contains one or more metanotions, then for some terminal production of such a metanotion, a new rule is obtained by replacing that metanotion, throughout a copy of the considered rule, by that terminal production, whereupon this new rule is considered instead and Step 3 is taken; otherwise, all blanks in the considered rule are removed and the rule so obtained is a production rule of the strict language.

b) A number of production rules may be added for 'indicant', 'dyadic indicant' and 'monadic indicant' (4.2.1.b,e,f), each of whose direct productions is a symbol different from any symbol given in this Report, with the restriction that no terminal production of 'indicant' is also a terminal production of 'monadic indicant'.

c) A number of production rules may be added for 'other comment item' (3.0.9.c) and 'other string item' (5.1.1.b) each of whose direct productions is a symbol different from any terminal production of 'character token' with the restrictions that no terminal production of 'other comment item' is 'comment symbol' and no terminal production of 'other string item' is 'quote symbol'.

(The rule
'actual LOWER bound : strict LOWER bound, flexible symbol option.'
derived from 7.1.1.t by Step 1 and considered in Step 2 is used in Step 3 to provide either one of the following two production rules of the strict language:
'actuallowerbound:strictlowerbound,flexiblesymboloption.' and
'actualupperbound:strictupperbound,flexiblesymboloption.';
however, to ease the burden on the reader, who may more easily ignore blanks himself, some blanks will be retained in the symbols, notions and production rules in the rest of this Report. Thus, the rules will be written in the more readable form
'actual lower bound : strict lower bound, flexible symbol option.' and
'actual upper bound : strict upper bound, flexible symbol option.'.
Note that
'actual lower bound : strict upper bound, flexible symbol option.'
is not a production rule of the strict language, since the replacement of
1.1.5. continued

the metanotion "LOWER" by one of its productions must be consistent throughout.
Since some metanotions have an infinite number of terminal productions, the number of notions in 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 1.1.1 and 8.5.2.2. These infinities should not worry the reader. From the sequel it follows that in any program only a finite number of notions and production rules are involved, and that although notions of infinite length may be involved, their constitution is always determined by a simple substitution process defined by a finite number of symbols, viz., the program; it is this process rather than its hypothetical outcome that matters.
Some production rules obtained from a rule containing a metanotion may be blind alleys in the sense that no production rule is given for some member to the right of the colon even though it is not a symbol.)

1.1.6. The semantics of the strict language

a) A terminal production of a notion is considered as a linearly ordered sequence of symbols. This order is termed 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 (but see 3.1.2.b).

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

c) A "parenition" when not in a section whose heading is, or begins with, "Syntax", not between "apostrophe's" ("'") and not contained in another paranotion "denotes" some number of protonotions, its "originals". A paranotion is either

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

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

iii) a notion whose production rule does (rules do) begin with an asterisk, and then it denotes any of its direct productions (which, in this Report, always is a notion or a symbol, e.g., "transcript" (8.5.1.1.j) denotes "trimmer" or "subscript"), or

iv) a paranotion in which one or more "hyphen's" ("-"") have been inserted and then it denotes the originals of that paranotion before the inserting (e.g., "begin symbol" denotes what "begin symbol" denotes), or

v) a paranotion followed by "s" or a paranotion ending with "y" in which that "y" has been replaced by "ies" and then it denotes some number of the original of that paranotion before the modification (e.g., "transcripts" denotes some number of "trimmer's" and/or "subscript's" and "primaries" denotes some number of the notions denoted by "primary"), or

vi) a paranotion whose first small syntactic mark has been replaced by the corresponding large syntactic mark, and then it denotes the originals of that paranotion before the modification (e.g., "Identifiers" denotes the notions denoted by "Identifiers"), or
vii) a paranotion in which a terminal production of 'SORT' and/or of 'SOME' and/or of 'MOID' has been omitted, and then it denotes the original of any other paranotion from which the given paranotion could be obtained by omitting a terminal production of 'SORT' and/or of 'SOME' and/or of 'MOID' (e.g., "Jump" denotes the notions denoted by "MOID jump" (8.2.7.1.c), "declaration" denotes the notions denoted by "SOME declaration" (6.2.1.a, 7.0.1.a) and "clause" denotes the notions denoted by "SORTITY SOME MOID clause" (6.1.1.a, 6.2.1.b, c, d, f, 6.3.1.a, 6.4.1.a, c, d, e, 8.1.1.a), where "SORTITY" ("SOME", "MOID") stands for any terminal production of the meta-notion "SORTITY" ("SOME", "MOID").

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

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

e) An occurrence $O$ of a terminal production of a notion $N$ is produced by a tree of specific productions; for this "production tree" are defined:
   a "direct descendent" of $N$, i.e., a member of the direct production of $N$;
   a "descendent" of $N$, i.e., a direct descendent of either $N$ or a descendent of $N$;
   the "offspring" of a descendent $O$ of $N$, i.e., if $O$ is a notion (symbol),
   then the occurrence of the terminal production of (the occurrence of) $O$
   (which is, or is contained in, $O$); and
   a "direct constituent" of $O$, i.e., the offspring of a direct descendent of $N$.

f) A paranotion $P$ which is said to be a direct constituent of a paranotion $Q$, stands for all occurrences of symbols denoted by $P$ or of terminal productions of notions denoted by $P$ which are direct constituents of occurrences of terminal productions of notions stood for by $Q$.

A paranotion $P$ is a descendent of a paranotion $Q$ if it is a direct constituent of $Q$ or if it is a direct constituent of a descendent of $Q$.

A paranotion $P$ is a "constituent" of a paranotion $Q$ if it is a descendent of $Q$ but not a descendent of a descendent $R$ of $Q$ where $R$ is the same protonotion as either $P$ or $Q$.

(Hence, $I : Z$ is a constituent actual-row-of-rower of the actual-declarer $[I : Z]$ struct $([3 : d]$ real $a)$, but $3 : d$ is not, since it is a descendent of the descendent actual-declarer $[3 : d]$ real. Also, $j := 1$ is a constituent assignation (8.3.1.1.a) of the assignation $i := j := 1$, but not of the serial-clause $i := j := 1$; $k := 3$ nor of the assignations $j := 1$ and $k := i := j := 1$. The assignation $j := 1$ is not a direct constituent of the assignation $i := j := 1$, but the source $j := 1$ is (8.3.1.1.a).)

g) A paranotion which is a direct constituent of a second paranotion is a paranotion of that second paranotion.
1.1.6. continued 2

(This permits the abbreviation of "direct constituent of", which would
appear frequently under Semantics, to "of", "its" or even "the", e.g., in
the assignment (8.3.1.1.a) \( i := j \); \( i \) is its destination or \( i \) is the, or a,
destination of the assignment \( i := j \); whereas \( i \) is a constituent, but not
simply a, destination of the serial-clause \( i := j; j := k \).)

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

i) If an occurrence of a sequence of symbols is the offspring of a direct
descendent \( D \) of a notion \( N \) which is the only member of a direct production of
\( N \), then its "preelaboration", possibly yielding a "prevalue" with a "pre-
mode" and a "prescope", as terminal production of \( N \) is its elaboration, pos-
sibly yielding a "value" with a "mode" and a "scope", as terminal production of
\( D \) and, except as otherwise specified (8.2), its elaboration with value,
mode and scope as terminal production of \( N \) is its preelaboration with pre-
value, premode and prescope as terminal production of \( N \).

(The elaboration with value, mode and scope of the reference-to-real-
confrontation (8.3.0.1.a) \( x := 3.1d \) is its preelaboration with prevalue,
premode and prescope which is its elaboration with value, mode and scope as a
reference-to-real-assignment.

The syntax of the strict language has been chosen in such a way that a
given sequence of symbols which is a terminal production of 'program' is so
by means of a unique set of productions, except, possibly, for production
rules inducing at most preelaboration, e.g., derived from rules 6.1.1.g,
6.2.1.e and 6.4.1.d (balancing of modes) and from rule 7.1.1.cc in combina-
tion with 7.2.1.a and 7.4.1.a (order of terminal productions of 'MODE' in a
terminal production of 'UNIFIED'); see also 2.3.a.)

j) A terminal production of a metanotion \( M \) is "enveloped" by a notion \( N \) if
it is contained once in \( N \) but not in another terminal production of \( M \) con-
tained in \( N \).

(Thus, 'reference to real' is enveloped as terminal production of 'MODE'
by 'reference to real mode identifier', but 'real' is not.)

k) If something is left "undefined" or is said to be undefined, then this
means that it is not defined by this Report alone, and that, for its defini-
tion, 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.

(Thus, real \( x \) is a representation of an identity-declaration in the ex-
tended language which means the same as real \( x := \log \) real which is a repre-
sentation of that identity-declaration in the strict language; see 9.2.a.)
1.1.8. The representation language

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

b) Each version of the language in which representations are used which are sufficiently close to the given representations to be 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, 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', 'BEGIN and 'BEGIN' (3.1.2.b) are all representations of the begin-symbol (3.1.1.e) in the representation language and some combination of holes in a punched card may be a representation of it in some hardware language.}

1.2. The metaproduction rules

1.2.1. Metaproduction rules of modes

a) MODE : MOOD ; UNITED.
b) MOOD : TYPE ; STOWED.
c) TYPE : PLAIN ; format ; PROCEDURE ; reference to MODE.
d) PLAIN : INTREAL ; boolean ; character.
e) INTREAL : INTEGER ; REAL.
f) INTEGER : LONGESTY integral.
g) REAL : LONGESTY real.
h) LONGESTY : long LONGESTY ; EMPTY.
i) EMPTY : .
j) PROCEDURE : procedure PARAMETY MOOD.
k) PARAMETY : with PARAMETERS ; EMPTY.
l) PARAMETERS : PARAMETER ; PARAMETERS and PARAMETER.
m) PARAMETER : MODE parameter.

n) MOOD : MODE ; void.
o) STOWED : structured with FIELDS ; row of MODE.
p) FIELDS : FIELD ; FIELDS and FIELD.
q) FIELD : MODE field TAG.
r) TAG : LETTER ; TAG LETTER ; TAG DIGIT.
s) LETTER : letter ALPHA ; letter aleph.
t) ALPHA : a ; b ; c ; d ; e ; f ; g ; h ; i ; j ; k ; l ; m ; n ; o ; p ;
q ; r ; s ; t ; u ; v ; w ; x ; y ; z.
u) DIGIT : digit FIGURE.
v) FIGURE : zero ; one ; two ; three ; four ; five ; six ; seven ; eight ;
nine.
w) UNITED : union of LMOODS MOOD mode.
x) LMOODS : MOOD ; LMOODS LMOOD.
y) LMOOD : MOOD and.

{The reader might find it helpful to note that a metanotion ending with 'ETY' always has 'EMPTY' as a production.}
1.2.2. Metaproduction rules associated with modes

a) PRIMITIVE : integral ; real ; boolean ; character ; format.
b) ROWS : row of ; ROWS row of.
c) ROWSETY : ROWS ; EMPTY.
d) ROWSEETY : ROWSETY.
e) NONROW : NONSTORED ; structured with FIELDS.
f) NONSTORED : TYPE ; UNITED.
g) REFETY : reference to ; EMPTY.
h) NONPROC : PLAIN ; format ; procedure with PARAMETERS MOID ; reference to NONPROC ; structured with FIELDS ; row of NONPROC ; UNITED.
i) FRAM : procedure with LMODE parameter and RMODE parameter MOID ; procedure with RMODE parameter MOID.
j) LMODE : MODE.
k) RMODE : MODE.
l) IMODE : MOOD and.
m) IMODESETY : MOOD and IMODESETY ; EMPTY.
n) RMODESETY : RMODESETY and MOOD ; EMPTY.
o) LSETY : IMODESETY.
p) BOX : IMODESETY box.
q) LFIELDSETY : FIELDS and ; EMPTY.
r) RFIELDSETY : and FIELDS ; EMPTY.
s) COMPLEX : structured with real field letter r letter e and real field letter i letter m.
t) BITS : structured with row of boolean field LENGTHETY letter aleph.	u) LENGTHETY : LENGTH LENGTHETY ; EMPTY.
v) LENGTH : letter l letter o letter n letter g.
w) BYTES : structured with row of character field LENGTHETY letter aleph.
x) STRING : row of character ; character.
y) MABEL : MODE mode ; label.

1.2.3. Metaproduction rules associated with phrases and coercion

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

1.2.4. Metaproduction rules associated with coercends

a) COERCEND : MOID FORM.
b) FORM : confrontation ; FORESE.
c) FORESE : ADIC formula ; cohesion ; base.
d) ADIC : PRIORITY ; monadic.
e) PRIORITY : priority NUMBER.
f) NUMBER : one ; TWO ; THREE ; FOUR ; FIVE ; SIX ; SEVEN ; EIGHT ; NINE.
1.2.4. continued

g) TWO : one plus one.
h) THREE : TWO plus one.
i) FOUR : THREE plus one.
j) FIVE : FOUR plus one.
k) SIX : FIVE plus one.
l) SEVEN : SIX plus one.
m) EIGHT : SEVEN plus one.
n) NINE : EIGHT plus one.

1.2.5. Other metaproduction rules

a) VIRTUAL : VIRTUAL; formal.
b) VIRACT : virtual; actual.
c) LOWER : lower; upper.
d) ANY : KIND; suppressible KIND; replicatable KIND;
replicable suppressible KIND.
e) KIND : sign; zero; digit; point; exponent; complex; string;
character.
f) NOTION : ALPHA; NOTION ALPHA.
g) SEPARATOR : LIST separator; go on symbol; completer; sequencer.
h) LIST : list; sequence.
i) PACK : pack; package.

{Rule f implies that all protonotions (1.1.2b) are productions
(1.1.3e) of the metanotion (1.1.3b) NOTION; for the use of this meta-
notion see 3.0.1.b,c,d,g,n,i.)

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

1.3. Pragmatics

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

(The rules under Syntax are provided with cross-references to be inter-
preted as follows. Let a "hypernotion" be either a protonotion or a sequence
of one or more metanotions, possibly preceded and/or separated and/or fol-
lowed by protonotions; then each rule consists of a hypernotion followed by
a colon followed by one or more hypernotions separated by commas or semico-
lons, and is closed by a point. By virtue of 1.1.5.a. Step 3, each hyperno-
tion eventually yields one or more protonotions. In each rule, a hypernotion
before (after) the colon is followed by indicators of the rules in which a
hypernotion yielding one or more protonotions also yielded by the first hy-
pernotion appears after (before) the colon, or by indicators of the repre-
sentations in section 3.1.1 of the symbols yielded by the first hypernotion.
Here, an indicator is, in principle, the section number followed by the let-
ter indicating the line where the rule or representation appears, with the
1.3. continued

following conventions:
  i) the indicators whose section number is that of the section in which
      they appear, are given first and their section number is omitted; e.g.,
      "3.0.3.b" appears as "b" in section "3.0.3";
  ii) all points are omitted and 10 appears as A; e.g., "3.0.3.a" appears as
       "303a" elsewhere and "3.0.10.a" appears as "304a";
  iii) a final 1 is omitted; e.g., "811a" appears as "81a";
  iv) a section number which is the same as in the preceding indicator is
       omitted; e.g., "821a, 821b" appears as "821a,b";
  v) numerous indicators of the rules 3.0.1.b up to i are replaced by more
      helpful indicators; e.g., in 6.1.1.c, "chain of strong void units
      separated by go on symbols[30c]" appears as "chain of strong void units[e]
      separated by go on symbols[31f]"; also, indicators in section 3.0.1 are
      restricted to a bare minimum;
  vi) the absence of a production rule for one or more protonations which are
      not symbols and are yielded by the hypernotion appearing after the colon,
      is indicated by "="; e.g., in 8.3.0.1.a after "MODE conformity relation"
      appears {832a,-} since 8.3.2.1.a yields only production rules for "boolean
      conformity relation", and no other rule provides the absent productions.

(Some of the pragmatic remarks are examples in the representation
language. In these examples, identifiers occur out of context from their defining occurrences. Unless otherwise specified, these occurrences identify those in the standard-precule (2.1.b and Chapter 10) (e.g., see 10.3.k for random and 10.3.a for pt), that in the exit (2.1.e) (viz., exit), or those in:

int i, j, k, m, n; real a, b, x, y; bool p, q, overflow; char c; format f;
bytes r; string s; bit t; compl w, z; ref real xx, yy; [1:n] real xl, y1;
[1:m, 1:n] real xz; [1:n, 1:n] real y2; [1:n] int i1; [1:m, 1:n] int i2;
proc ref real x or y = ref real : (random < .5 | x | y));
proc noam = (int i) real : cos (2 * pi * i / n);
proc ptsn = (int i) real : sin (2 * pi * i / n);
proc y = (real u) real : (arctan (w - a + u - 1));
proc stop = go to exit;
prinon;.

Merely corroborative detail, intended to give artistic verisimilitude to an otherwise bald and unconvincing narrative.
Mikado,

W.S. Gilbert.)
2. The computer and the program

(The programmer is concerned with particular-programs (2.1.d). These are always contained in a program (2.1.a), which also contains the standard-prelude, i.e., a declaration-prelude which is always the same (see Chapter 10), possibly a library-prelude, i.e., a declaration-prelude which may depend upon the implementation, the exit, i.e., ; exit:, which enables the programmer to end the elaboration of a program anywhere by that of the jump exit, possibly a library-postlude, and the standard-postlude (10.5).)

2.1. Syntax

a) program : open symbol(31e), standard prelude(b),
   library prelude(c) option, particular program(d), exit(e),
   library postlude(f) option, standard postlude(g), close symbol(31e).
   b) standard prelude(a) : declaration prelude(61b) sequence.
   c) library prelude(a) : declaration prelude(61b) sequence.
   d) particular program(a) :
      label(61k) sequence option, strong CLOSED void clause(62b,63a,64a).
   e) exit(a) : go on symbol (31f),
      letter e letter x letter i letter t(41c), label symbol(31e).
   f) library postlude(a) : statement interlude(61i).
   g) standard postlude(a) : strong void clause train(61h).

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 symbols or of terminal productions (1.1.2.f) of actions. Internal objects are "instances" of values (2.2.3).

2.2.2. Relationships

a) Relationships either are "permanent", i.e., independent of the program and its elaboration, or actions may cause them to hold or cease to hold. Each relationship is either between external objects or between an external object and an internal object or between internal objects.

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

c) A given occurrence of a terminal production of 'MABEL identifier' (4.1.1.b) ('MODE mode indication' (4.2.1.b) or 'PRIORITY indication' (4.2.1.e), 'PRAM/ADIC operator' (4.3.1.b,c)) where 'MABEL' ('MODE', 'PRIORITY', 'PRAM', 'ADIC') stands for any terminal production of the meta-notion 'MABEL' ('MODE', 'PRIORITY', 'PRAM', 'ADIC') may identify a "defining occurrence" ("indication-defining occurrence", "operator-defining occurrence") of the same terminal production.

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

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

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

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

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

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

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

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

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

Values are "plain values" [2.2.3.1], "structured values" [2.2.3.2], "multiple values" [2.2.3.3], routines [2.2.3.4], "formats" [2.2.3.4], and names [2.2.2.1, 2.2.3.5].

2.2.3.1. Plain values

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

b) An arithmetic value has a "length number", i.e., a positive integer characterizing the degree of discrimination with which the value is kept in the computer. The number of integers (real numbers) of given length number that can be distinguished increases with the length number up to a certain length number, the number of different lengths of integers (real numbers) (10.1.a,c), after which it is constant.

c) For each pair of integers (real numbers) of the same length number, the relationship "to be smaller than" is defined (10.2.3.a, 10.2.4.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 (10.2.3.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 (10.2.4.g,1,m); these real numbers are obtained "in the sense of numerical analysis", i.e., by performing the operations known in mathematics by these terms on real numbers which may deviate slightly from the given ones (; this deviation is left undefined in this Report).

d) Each integer of given length number is equivalent to a real number of that length number. Also, each integer (real number) of given length number is equivalent to an integer (real number) whose length number is greater by one. These equivalences permit the "widening" (8.2.5) of an integer into a real number and the increase of the length number of an integer or real number (10.2.3.q, 10.2.4.n). The inverse transformations are only possible on those real numbers (arithmetic values) which are equivalent to an integer (10.2.4.p) (a value of smaller length number (10.2.3.r, 10.2.4.q)).

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

f) Each character is equivalent to a nonnegative integer of length number one, its "integral equivalent" (10.1.j); this relationship is defined only to the extent that different characters have different integral equivalents.

2.2.3.2. Structured values

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

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

b) The descriptor consists of an "offset", \(c\), and some number, \(n > 0\), of "quintuples" \(\{\ell_i, u_i, d_i, s_i, t_i\}\) of integers, \(i = 1, \ldots, n\); \(\ell_i\) is the \(i\)-th "lower bound", \(u_i\) the \(i\)-th "upper bound", \(d_i\) the \(i\)-th "stride", \(s_i\) the \(i\)-th "lower state" and \(t_i\) the \(i\)-th "upper state". If for any \(i\), \(i = 1, \ldots, n\), \(u_i < \ell_i\), then the number of elements in the multiple value is zero; otherwise, it is \((u_1 - \ell_1 + 1) \times \cdots \times (u_n - \ell_n + 1)\). The descriptor "describes" each element selected by \(c + (r_1 - \ell_1) \times d_1 + \cdots + (r_n - \ell_n) \times d_n\) where \(\ell_i \leq r \leq u_i\) for each \(i = 1, \ldots, n\).

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

c) A subvalue of a given multiple value is a multiple value which is referred to by the value of a slice \(\{8.3.1.1.1\}\) the value of whose primary refers to the given multiple value.

d) If each element of an instance \(I\) of a multiple value \(M\) is the same instance of a value as the corresponding element of an instance \(J\) of \(M\), then \(I\) and \(J\) are one same instance of \(M\).

2.2.3.4. Routines and formats

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

2.2.3.5. Names

a) There is one name, \(\text{nil}\), 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 \(\{7.1.2.4.\text{Step 6}\}\), a rowed-coercer \(\{8.2.6.2.\text{Step 7}\}\) or a skip \(\{8.2.7.2.\}\) \{(\), and refers to precisely one instance of a value).

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

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

d) If an instance of a name and an instance of a second name refer to one same instance of a value, then they are instances of one same name.
2.2.4. Modes and scopes

2.2.4.1. Modes

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

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 \(\lfloor n - 1 \rfloor\) times 'long' followed by 'integral' (by 'real').

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

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

f) The mode of a routine is a terminal production of 'PROCEDURE' (1.2.1.j).

g) The mode of a name is 'reference to' followed by another mode; if the name is not nil, then that other mode is either the mode of the value to which the name refers, or is united from (4.4.3.a) it. (See 7.1.2.d Step 8.)

2.2.4.2. Scopes

a) Each value has one specific scope.

b) The scope of a plain value is the program, that of a structured (multiple) value is the smallest of the scopes of its fields (elements, if any, and, otherwise, the program), that of a routine or format possessed by a given denotation D (5.4.1.a, 5.5.1.a) (of a routine which is the same sequence of symbols as a given cast-pack D or void-cast-pack-pack D (8.3.4.1.a)) is the smallest range {4.1.1.e} containing a defining occurrence (4.1.2.a) (indication-defining occurrence (4.2.2.a), operator-defining occurrence (4.3.2.a)) of a terminal production of a notion ending with 'identifier' ('indication', 'operator'), if any, not contained in D but identified (4.1.2.b (4.2.2.b, 4.3.2.b)) by some applied (indication-applied, operator-applied) occurrence of that terminal production contained in D, and, otherwise, the program, and that of a name is some (2.2.3.5, 8.5.1.2.b) range.
2.2.5. Actions

(Suit the action to the word, the word to the action.
Hamlet, William Shakespeare.)

a) An action is "inseparable", "serial" or "collateral". A serial action consists of actions which take place one after the other.

b) A collateral action consists of actions merged in time; i.e., it consists of inseparable actions each of which is chosen, in a way which is left undefined in this Report, from amongst the first of the inseparable actions which, at that moment, according to this Report, would be the continuation of any of the constituting actions.

c) The elaboration of any (of the closed-clause following the do-symbol (3.1.1.h) in any) closed-clause (6.3.1.a) which is a modified copy (8.4.2) of the actual-parameter of the operation-declaration (7.5.1.a) 10.4.d (10.4.c) is an inseparable action.

(What other actions are inseparable is left undefined in this Report.)

2.3. Semantics

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

a) The elaboration of a program is the elaboration of the strong-closed-void-clause (6.3.1.a) consisting of the same sequence of symbols.

(In this Report, the syntax says which sequences of symbols are programs and the semantics which actions are performed by the computer when elaborating a program. Both syntax and semantics are recursive. Though certain sequences of symbols may be terminal productions of 'program' in more than one way (1.1.6.i), this syntactic ambiguity does not lead to a semantic ambiguity.)

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

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

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

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

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

\{pr algol 68 pr
    begin
    proc pr nonrec pr p = (; p);
    p
    end
    pr run pr pr ? pr
report on the Algorithmic
Language ALGOL 68.\}
3. Basic tokens and general constructions

3.0. Syntax

3.0.1. Introduction

a)* basic token : letter token[302a] ; denotation token[303a] ;
   action token[304a] ; declaration token[305a] ;
   syntactic token[306a] ; sequencing token[307a] ;
   hip token[308a] ; extra token[309a] ; special token[30Aa].
  \*b) NOTION option : NOTION ; EMPTY.
  \*c) chain of NOTIONS separated by SEPARATORS(c,d) : NOTION ;
      NOTION, SEPARATOR(e,f,31f,61j,1),
      chain of NOTIONS separated by SEPARATORS(c).
  \*d) NOTION LIST(g) : chain of NOTIONS separated by LIST separators(c,e,f).
  \*e) list separator(c,d,g) : comma symbol[31e] .
  \*f) sequence separator(c,d,g) : EMPTY .
  \*g) NOTION LIST proper : NOTION, LIST separator(e,f), NOTION LIST(d).
  \*h) NOTION pack : open symbol[31e], NOTION, close symbol[31e].
  \*i) NOTION package : begin symbol[31e], NOTION, end symbol[31e].

{Examples:
  a) a ; 0 ; + ; int ; if ; ; nil ; for ; " ;
  b) 0 ; ; (integral-part-options)
  c) 0,1,2 ; (a chain-of-strong-integral-units-separated-by-list-separators)
  d) 0 , 0, 1, 2 ; (strong-integral-unit-lists)
  e) , ;
  g) 1, 2, 3 ; (a strong-integral-unit-list-proper)
  h) (1, 2, 3) ; (a strong-integral-unit-list-proper-pack)
  i) begin w := 8.1e; y := 8.7e end (a strong-serial-void-clause-package) }

3.0.2. Letter tokens

a)* letter token : LETTER(b).
  b) LETTER(3094, 41b, c, d, 512b, 45b, i, o, q, 552b, e, f, 553f, 554a, 555b, 556c, 557c,
       71j) : LETTER symbol[31a].

{Examples:
  a) a ; (see 1.1.4, Step 2)}

 {Letter-tokens either are, or are constituents of, identifiers
  (4.1.1.a), field-selectors (7.1.1.i), real-denotations (5.1.2.1.a), format-
  denotations (5.5.1.a) and string-items (5.3.1.d).}

3.0.3. Denotation tokens

a)* denotation token : number token(b) ; true symbol[31b] ;
   false symbol[31b] ; formatter symbol[31b] ; flipflop(e) ;
   space symbol[31b].
  b) number token[3094d] ; digit token(c) ; point symbol[31b] ;
     times ten to the power symbol[31b].
  c) digit token(b, 511a) : DIGIT(4).
  d) DIGIT(c, 514a, 552e) : DIGIT symbol[31b].
  e) flipflop[3094d, 52e] : flip symbol[31b] ; flop symbol[31b].
3.0.3. continued

{Examples:
  a) \texttt{1; \texttt{true}; \texttt{false}; \#; \#; 1; \#;}
  b) \texttt{1; \#; 10;}
  c) \texttt{1;}
  d) \texttt{1;}
  e) \texttt{1; 0;}
}

(Denotation-tokens are, or are constituents of, denotations (5.0.1.a). Some denotation-tokens may, by themselves, be denotations, e.g., the digit-token 1, whereas others, e.g., the formatter-symbol \#, serve only to construct denotations.)

3.0.4. Action tokens

a)* action token: operator token(b); equals symbol(31c); times symbol(31c); confrontation token(d).

b) operator token(42e,f): minus and becomes symbol(31c); plus and becomes symbol(31c); times and becomes symbol(31c); divided by and becomes symbol(31c); over and becomes symbol(31c); modulo and becomes symbol(31c); plus and becomes symbol(31c); or symbol(31c); and symbol(31c); differs from symbol(31c); is less than symbol(31c); is at most symbol(31c); is at least symbol(31c); is greater than symbol(31c); plusminus(c); divided by symbol(31c); over symbol(31c); modulo symbol(31c); th element of symbol(31c); lower bound of symbol(31c); upper bound of symbol(31c); lower state of symbol(31c); upper state of symbol(31c); plus i times symbol(31c); not symbol(31c); down symbol(31c); up symbol(31c); absolute value of symbol(31c); binal symbol(31c); representation of symbol(31c); lengthen symbol(31c); shorten symbol(31c); odd symbol(31c); sign symbol(31c); round symbol(31c); real part of symbol(31c); imaginary part of symbol(31c); conjugate of symbol(31c); booleans to bits symbol(31c); characters to bytes symbol(31c).

c) plusminus(b,52i,55p): plus symbol(31c); minus symbol(31c).

d)* confrontation token: becomes symbol(31c); conforms to symbol(31c); conforms to and becomes symbol(31c); is symbol(31c); is not symbol(31c); cast of symbol(31c).

{Examples:
  a) \texttt{+; \#; \#; 1; \#;}
  b) \texttt{\#; \#; 10;}
  c) \texttt{1; \#; 10;}
  d) \texttt{1;}
  e) \texttt{1; 0;}
}

(Operator-tokens are constituents of formulas (8.4.1.a). Confrontation-tokens are constituents of confrontations (8.3.0.1.a).}
3.0.5. Declaration tokens

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

{Examples:
  a) int ; long ; struct ; ref ; flex ; either ; proc ; union ; mode ;
     compl ; byte ; bytes ; string ; sema ; file ; priority ; loc ; heap ;
     op
}

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

3.0.6. Syntactic tokens

a) syntactic token : open symbol(31e) ; close symbol(31e) ;
begin symbol(31e) ; end symbol(31e) ; comma symbol(31e) ;
parallel symbol(31e) ; sub symbol(31e) ; bus symbol(31e) ;
up to symbol(31e) ; at symbol(31e) ; if symbol(31e) ;
THENCE symbol(31e) ; fi symbol(31e) ; of symbol(31e) ;
label symbol(31e).

{Examples:
  a) ( ; ) ; begin ; end ; ; par ; [ ; ] ; ; ; if ; then ; fi ; of ; : }

(Syntactic-tokens separate external objects or group them together.)

3.0.7. Sequencing tokens

a) sequencing token : go on symbol(31f) ; completion symbol(31f) ;
go to symbol(31f).

{Examples:
  a) ; ; ; go to
}

(Sequencing-tokens are constituents of clauses, in which they specify
the order of elaboration (6.1.1.b,c,i,l, 8.2.7.1.c).)

3.0.8. Hip tokens

a) hip token : skip symbol(31g) ; nil symbol(31g).

{Examples:
  a) skip ; nil
}

(Hip-tokens function as skips and nihil (8.2.7.1.b,d).)
3.0.9. Extra tokens and comments

a) extra token: for symbol{31h} ; from symbol{31h} ; by symbol{31h} ;
to symbol{31h} ; while symbol{31h} ; do symbol{31h} ;
THESE if symbol{31h}.
b) comment(9.1) : comment symbol{31i} , comment item(c) sequence option,
comment symbol{31i}.
c) comment item(b) : character token(d) ; other comment item(1.1.5.c).
d) character token(c,514b) : LETTER(302b) ; number token(303b) ;
flipflop(303e) ; plus i times symbol{31e} ; open symbol{31e} ;
close symbol{31e} ; comma symbol{31e} ; space symbol{31b}.

{Examples:}
a) for ; from ; by ; to ; while ; do ; thef ;
b) \( \varphi \) with respect to \( \varphi \);
c) \( \nu ; \nu ; \nu ; \nu \);
d) \( a ; i ; i ; i ; ( i ) ; , ; , \)

3.0.10. Special tokens

a) special token: quote symbol{31i} ; comment symbol{31i} ;
indicant(1.1.5.b) ; dyadic indicant(1.1.5.b) ;
monadic indicant(1.1.5.b).

{Examples:}
a) " ; \( \varphi \) ; primitive ; \( \nu ; \nu \) 

3.1. Symbols

3.1.1. Representations

a) Letter tokens

<table>
<thead>
<tr>
<th>symbol</th>
<th>representation</th>
<th>symbol</th>
<th>representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter a</td>
<td>symbol{302b}</td>
<td>a</td>
<td>letter n</td>
</tr>
<tr>
<td>letter b</td>
<td>symbol{302b}</td>
<td>b</td>
<td>letter o</td>
</tr>
<tr>
<td>letter c</td>
<td>symbol{302b}</td>
<td>c</td>
<td>letter p</td>
</tr>
<tr>
<td>letter d</td>
<td>symbol{302b}</td>
<td>d</td>
<td>letter q</td>
</tr>
<tr>
<td>letter e</td>
<td>symbol{302b}</td>
<td>e</td>
<td>letter r</td>
</tr>
<tr>
<td>letter f</td>
<td>symbol{302b}</td>
<td>f</td>
<td>letter s</td>
</tr>
<tr>
<td>letter g</td>
<td>symbol{302b}</td>
<td>g</td>
<td>letter t</td>
</tr>
<tr>
<td>letter h</td>
<td>symbol{302b}</td>
<td>h</td>
<td>letter u</td>
</tr>
<tr>
<td>letter i</td>
<td>symbol{302b}</td>
<td>i</td>
<td>letter v</td>
</tr>
<tr>
<td>letter j</td>
<td>symbol{302b}</td>
<td>j</td>
<td>letter w</td>
</tr>
<tr>
<td>letter k</td>
<td>symbol{302b}</td>
<td>k</td>
<td>letter x</td>
</tr>
<tr>
<td>letter l</td>
<td>symbol{302b}</td>
<td>l</td>
<td>letter y</td>
</tr>
<tr>
<td>letter m</td>
<td>symbol{302b}</td>
<td>m</td>
<td>letter z</td>
</tr>
</tbody>
</table>

{No representation for 'letter aleph symbol' is provided and the pro-
grammer cannot provide one himself; see 1.1.4. Step 2, 3.1.2.c.)
3.1.1. continued

b) Denotation tokens

<table>
<thead>
<tr>
<th>symbol</th>
<th>representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>digit zero symbol{303a}</td>
<td>0</td>
</tr>
<tr>
<td>digit one symbol{303a,73b}</td>
<td>1</td>
</tr>
<tr>
<td>digit two symbol{303a,73c}</td>
<td>2</td>
</tr>
<tr>
<td>digit three symbol{303a,73d}</td>
<td>3</td>
</tr>
<tr>
<td>digit four symbol{303a,73e}</td>
<td>4</td>
</tr>
<tr>
<td>digit five symbol{303a,73f}</td>
<td>5</td>
</tr>
<tr>
<td>digit six symbol{303a,73g}</td>
<td>6</td>
</tr>
<tr>
<td>digit seven symbol{303a,73h}</td>
<td>7</td>
</tr>
<tr>
<td>digit eight symbol{303a,73i}</td>
<td>8</td>
</tr>
<tr>
<td>digit nine symbol{303a,73j}</td>
<td>9</td>
</tr>
<tr>
<td>point symbol{303b,512a,553c}</td>
<td>.</td>
</tr>
<tr>
<td>times ten to the power symbol{303b,512h}</td>
<td>10</td>
</tr>
<tr>
<td>true symbol{513a}</td>
<td>true</td>
</tr>
<tr>
<td>false symbol{513a}</td>
<td>false</td>
</tr>
<tr>
<td>format symbol{55a}</td>
<td>$</td>
</tr>
<tr>
<td>flip symbol{303e}</td>
<td>1</td>
</tr>
<tr>
<td>flop symbol{303e}</td>
<td>0</td>
</tr>
<tr>
<td>space symbol{309d}</td>
<td>~</td>
</tr>
</tbody>
</table>

c) Action tokens

<table>
<thead>
<tr>
<th>symbol</th>
<th>representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>minus and becomes symbol{304b}</td>
<td>-=: minus</td>
</tr>
<tr>
<td>plus and becomes symbol{304b}</td>
<td>+=: plus</td>
</tr>
<tr>
<td>times and becomes symbol{304b}</td>
<td>*=: times</td>
</tr>
<tr>
<td>divided by and becomes symbol{304b}</td>
<td>/=: div</td>
</tr>
<tr>
<td>over and becomes symbol{304b}</td>
<td>*=:= overb</td>
</tr>
<tr>
<td>modulo and becomes symbol{304b}</td>
<td>%=:= modb</td>
</tr>
<tr>
<td>prus and becomes symbol{304b}</td>
<td>+=:= prus</td>
</tr>
<tr>
<td>or symbol{304b}</td>
<td>v</td>
</tr>
<tr>
<td>and symbol{304b}</td>
<td>&amp;</td>
</tr>
<tr>
<td>differs from symbol{304b}</td>
<td>≠</td>
</tr>
<tr>
<td>is less than symbol{304b}</td>
<td>&lt;</td>
</tr>
<tr>
<td>is at most symbol{304b}</td>
<td>≤</td>
</tr>
<tr>
<td>is at least symbol{304b}</td>
<td>≥</td>
</tr>
<tr>
<td>is greater than symbol{304b}</td>
<td>&gt;</td>
</tr>
<tr>
<td>divided by symbol{304b}</td>
<td>/</td>
</tr>
<tr>
<td>over symbol{304b}</td>
<td>↑</td>
</tr>
<tr>
<td>modulo symbol{304b}</td>
<td>∈</td>
</tr>
<tr>
<td>th element of symbol{304b}</td>
<td>∑</td>
</tr>
<tr>
<td>lower bound of symbol{304b}</td>
<td>l</td>
</tr>
<tr>
<td>upper bound of symbol{304b}</td>
<td>u</td>
</tr>
<tr>
<td>lower state of symbol{304b}</td>
<td></td>
</tr>
<tr>
<td>upper state of symbol{304b}</td>
<td>\</td>
</tr>
<tr>
<td>plus i times symbol{304b,309d}</td>
<td>+</td>
</tr>
<tr>
<td>not symbol{304b}</td>
<td>¬</td>
</tr>
<tr>
<td>down symbol{304b}</td>
<td>↓</td>
</tr>
<tr>
<td>up symbol{304b}</td>
<td>↑</td>
</tr>
<tr>
<td>absolute value of symbol{304b}</td>
<td>abs</td>
</tr>
<tr>
<td>binal symbol{304b}</td>
<td>bin</td>
</tr>
</tbody>
</table>
3.1.1. continued 2

symbol representation
representation of symbol[304b] repr
lengthen symbol[304b] leng
shorten symbol[304b] short
odd symbol[304b] odd
sign symbol[304b] sign
round symbol[304b] round
real part of symbol[304b] re
imaginary part of symbol[304b] im
conjugate of symbol[304b] conj
booleans to bits symbol[304b] bit
characters to bytes symbol[304b] oth
plus symbol[304c] +
minus symbol[304c] -
equals symbol[42e,72a,73a,74a,75a] eq

times symbol[42e] *
becomes symbol[831a]
conforms to symbol[832b]
conforms to and becomes symbol[832b]
is symbol[833b]
is not symbol[833b]
cast of symbol[834a]

d) Declaration tokens

symbol representation
integral symbol[71c] int
real symbol[71c] real
boolean symbol[71c] bool
character symbol[71c] char
format symbol[71a] format
long symbol[42e, e, f, 510b, 52b, 71d] long
structure symbol[71e] struct
reference to symbol[71l, m, n] ref
categorical symbol[71v, v] flex
either symbol[71v] either
procedure symbol[71w]
union of symbol[71e, i] union
mode symbol[72a]
complex symbol[42c] comp
bits symbol[42c] bits
string symbol[42c] string
sema symbol[42e] sema
file symbol[42e] file
priority symbol[73a]
local symbol[851b]
heap symbol[851c]
operation symbol[75b] op
3.1.1. continued 3

e) Syntactic tokens

symbol representation

open symbol\{2a,30h,309d,54b,554b\} ( \\
close symbol\{2a,30h,309d,54b,554b\} ) \\
begin symbol\{301\} begin \\
end symbol\{301\} end \\
comma symbol\{30e,309d,54d,554b,62e,571f,91e85,861b,c\} , \\
parallel symbol\{62b\} par \\
sub symbol\{71o,p,861a\} [ \\
bus symbol\{71o,p,861a\} ] \\
up to symbol\{71r,861f\} : .. \\
at symbol\{86ig\} \@ \at \\
if symbol\{64a\} \{ if \texttt{case} \\
then symbol\{64e\} \{ then \texttt{in} \\
ext else symbol\{64e\} \{ \texttt{else out} \\
gfi symbol\{64a\} \{ \texttt{fi} \texttt{eof} \\
of symbol\{852a\} + \texttt{of} \\
label symbol\{8e,61k\} : .. \\
f) Sequencing tokens

symbol representation

go on symbol\{2e,30c,54d,61b,c,j\} ; .. \\
completion symbol\{611\} \texttt{exit} \\
go to symbol\{827\} \texttt{go to} \\
g) Hip tokens

symbol representation

skip symbol\{827b\} \texttt{skip} \\
nil symbol\{827b\} \texttt{nil} \\
h) Extra tokens

symbol representation

for symbol\{9,3,a,b\} \texttt{for} \\
from symbol\{9,3,a,b,c\} \texttt{from} \\
by symbol\{9,3,a,b,c\} \texttt{by} \\
to symbol\{9,3,a,c\} \texttt{to} \\
while symbol\{9,3,a,b,c\} \texttt{while} \\
do symbol\{9,3,a,b,c\} \texttt{do} \\
then if symbol\{9,4,b\} \texttt{then if} \\
ext if symbol\{9,4,b,c\} \texttt{else if} \\
i) Special tokens

symbol representation

quote symbol\{514a,c,53b\} \texttt{quote} \\
comment symbol\{309b\} \# \texttt{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
\( a > b \text{ then } b \ | \ a \text{ else } a \text{ if} \)
though acceptable to an automaton, would be more intelligible to a human in either of the two representations
\( a > b \ | \ b \ | \ a \)
or
\( \text{if } a > b \text{ then } b \text{ else } a \text{ if} \).
Also, some representations may not be available in a given implementation.)

b) A representation which is a sequence of underlined or bold-faced marks or a sequence of marks preceded by a "bold-face shift" (e.g., apostrophe,) and ending on a "light-face shift" (e.g., any mark different from the representation of a letter or digit,) or a sequence of marks between apostrophes is different from the sequence of those marks when not underlined, bold-faced, preceded by a bold-face shift and ending on a light-face shift or between apostrophes.

c) Representations of other terminal productions of 'letter token'
\{1.1.4.Step 2\}, 'indicant', 'dyadic indicant', 'monadic indicant' \{1.1.5.b\},
'other comment item' and 'other string item' \{1.1.5.c\} may be added, pro-
vided that no sequence of representations of symbols can be confused with any other such sequence.

(e.g., do if are representations of a do-symbol followed by an if-symbol,
whereas do if might be an ill-chosen representation of an indicant.)

d) The fact that representations of the terminal productions of 'letter
token' as given in 3.1.1.a 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 another terminal production of 'letter token'
\{1.1.4.Step 2\}.

(For certain different symbols, one same or nearly the same repre-
sentation is given; e.g., for the cast-of-symbol, up-to-symbol and label-symbol
respectively, the representations ":", ":" and ":", and, moreover, for all
of them the representation ":." is given. It follows uniquely from the syn-
tax which of these three symbols is represented by an occurrence of any mark
similar to one of these representations 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 ":", the representation of the cast-of-symbol, etc., and ":=",
the representation of the equals-symbol, and the representation ":." of the
cast-of-symbol, etc., appears to consist of ":" and ":", each of which might
be the representation of a point-symbol or completion symbol. It follows
from the syntax that ":=" can occur outside comments and row-of-character-
denotations as representation of the becomes-symbol only (since ":=" cannot
occur as representation of a monadic-operator). Similarly, the other given
composite representations do not cause ambiguity.)
4. Identification and the context conditions

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

4.1. Identifiers

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

4.1.1. Syntax

a)* identifier : MABEL identifier(b).
b) MABEL identifier(\{a4e,61k,827c,860a\} : TAG(c,d,302b).
c) TAG LETTER(b,c,d,2e,71j) : TAG(c,d,302b), LETTER(302b).
d) TAG DIGIT(b,c,d,71j) : TAG(c,d,302b), DIGIT(303d).
e)* range : program(2a) ; SORTITY serial CLAUSE(61a) ;
    procedure with PARAMETERS MOID notation(54b).

(Examples:
 b) \( x \ ; \ xx \ ; \ x1 \ ; \ \text{amsterdam} \)

(Rule b together with 1.2.1.r and 1.2.2.y 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 mode identifier : letter a letter b.' is one such a production rule. From rules 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 mode identifier'. For additional insight into the function of rules c and d, see 7.1.1.j and 8.5.2.1.a.)

4.1.2. Identification of identifiers

(The method of identification of identifiers is first to distinguish between defining and applied occurrences and then to discover which defining occurrence is identified by a given applied occurrence.)
4.1.2. continued

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

b) If a given occurrence of a terminal production of 'MAEBL identifier' (see a) is an applied occurrence, then it may identify a defining occurrence of the same terminal production found by the following steps:
Step 1: The given occurrence is termed the "home" and Step 2 is taken;
Step 2: If there exists a smallest range containing the home, then this range, with the exclusion of all ranges contained within it, is termed the home and Step 3 is taken (; otherwise, there is no defining occurrence which the given occurrence identifies; see 4.4.1.b);
Step 3: If the home contains a defining occurrence of the same terminal production of 'MAEBL identifier', then the given occurrence identifies it; otherwise, Step 2 is taken.

{In the closed-clause (string s := "aba"; s [s] ≠ "d"), the first occurrence of s is a defining occurrence of a terminal production of 'reference to row of character mode identifier'. The second occurrence of s identifies the first and, in order to satisfy the identification condition (4.4.1.a), is also a terminal production of 'reference to row of character mode identifier'. Identifiers have no inherent meaning.}

4.2. Indications

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

4.2.1. Syntax

a) * indication : MODE mode indication(b) ; ADIC indication(e,f).
b) MODE mode indication(71b,ii,72a) : mode standard(c) ; indiant(1.1.5.b).
c) mode standard(b) :
   string symbol(31d) ; sema symbol(31d) ; file symbol(31d) ;
   long symbol(31d) sequence option, complex symbol(31d) ;
   long symbol(31d) sequence option, bits symbol(31d) ;
   long symbol(31d) sequence option, bytes symbol(31d).
d) * dyadic indication : PRIORITY indication(e).
e) PRIORITY indication(43b,73a) : dyadic indication(1.1.5.b) ;
   long symbol(31d) sequence option, operator token[304b] ;
   long symbol(31d) sequence option, equals symbol[31c] ;
   long symbol(31d) sequence option, times symbol(31e).
f) monadic indication(43c) : monadic indication(1.1.5.b) ;
   long symbol(31d) sequence option, operator token[304b].
g) * adic indication : ADIC indication(e,f).

{Examples:
   b) compl ; primitive ;
   c) string ; sema ; file ; long compl ; bite ; long bytes ;
   d) ? ; + ; = ; x ;
   f) ! ; + ; long bitb }
4.2.2. Identification of indications

(The identification of indications is similar to that of identifiers.)

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

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

"Step 3: If the home contains an indication-defining occurrence of the same terminal production of 'MODE mode indication' ('PRIORITY indication'), then the given occurrence identifies it; otherwise, Step 2 is taken."

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

4.3. Operators

(Operators are either monadic-operators, i.e., require a right operand only, or are dyadic-operators, i.e., require both a left and a right operand, e.g., \textit{abs} and / in \textit{abs} \textit{x} and \textit{x} / \textit{y}. Operators are made to possess routines by the elaboration of operation-declarations (7.5.1.a). Operators are identified by observing the modes of their operands, e.g., \textit{x + y}, \textit{x + i}, \textit{i + x}, \textit{i + j} each involves a different operator, see 10.2.4.1, 10.2.5.a, 10.2.5.b and 10.2.3.i. Though the mode enveloped by the original of an operator contains the mode of the value, if any, delivered by its routine, this mode is not involved in the identification process.)

4.3.1. Syntax

a\textasteriskcentered) operator : PRAM ADIC operator\textcommata b).

b) procedure with IMODE parameter and RMODE parameter MOID PRIORITY operator(75,4.4b) : PRIORITY indication\textcommata e2).

c) procedure with RMODE parameter MOID monadic operator(75,4.4g) : monadic indication\textcommata e2f).

d\textasteriskcentered) dyadic operator : procedure with IMODE parameter and RMODE parameter MOID PRIORITY operator(b).

e\textasteriskcentered) monadic operator : procedure with RMODE parameter MOID monadic operator(c).

{Examples:

b) + ;
c) abs }

4.3.2. Identification of operators

(The identification of operators is similar to that of identifiers and indications, except that different occurrences of one same terminal production of 'ADIC indication' may be occurrences of more than one terminal production of 'PRAM ADIC operator' and, therefore, the modes of the operands must be considered.)
A given occurrence of a terminal production of 'FRAM ADIC operator' where "FRAM" ("ADIC") stands for any terminal production of the metanotion 'FRAM' ("ADIC") is an operator-defining occurrence if it precedes the equals-symbol of an operation-declaration (7.5.1.a); otherwise, it is an "operator-applied occurrence".

b) If a given (operator-applied) occurrence of a terminal production of 'FRAM ADIC operator' (see a) is the operator of a formula F (8.4.1.a), then it may identify an operator-defining occurrence of the same terminal production found by using the steps of 4.1.2.b, with Step 3 replaced by:

"Step 3: If the home contains an operator-defining occurrence 0, (in an operation-declaration (7.5.1.a,b)) of a terminal production T of 'FRAM ADIC operator' which is the same terminal production of 'ADIC indication' as the given occurrence, and which (the identifications of all descendendent identifiers, indications and operators of the operand(s) of F having been made,) is such that some formula exists which is the same sequence of symbols as F, whose operator is an occurrence of T and which is such that the original of each descendendent identifier, indication and operator of its operand(s) is the same notion as the original of the corresponding identifier, indication and operator contained in F (which, if the program is a proper program, is uniquely determined by virtue of 4.4.1.a, then the given occurrence identifies 0; otherwise, Step 2 is taken."

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

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

In the closed-clause

\[
\text{begin real } x, y := 1.5; \text{ priority } \min = 6; \text{ op } \min = (\text{real } a, b) \text{ real } : (a > b | b \leq a); x := y \min pt / 2 \text{ end },
\]

the first occurrence of \(\min\) is an indication-defining priority-SIX-indication. The second occurrence of \(\min\) is indication-applied and identifies the first occurrence (4.2.2), whereas, at the same textual position, \(\min\) is also operator-defined as a [prtr]-priority-SIX-operator, where "[prtr]" stands for "procedure-with-real-parameter-and-real-parameter-real". The third occurrence of \(\min\) is indication-applied and, as such, identifies the first occurrence, whereas, at the same textual position, \(\min\) is also operator-applied, and, as such, identifies the second occurrence; this makes it, because of the identification condition (4.4.1.a), a [prtr]-priority-SIX-operator. This identification of the dyadic-operator is made because:

i) \(\min\) occurs in an operation-declaration,

ii) the base \(y\) can be firmly coerced to the mode specified by \textit{real},

iii) the formula \(pt / 2\) is a priori of the mode specified by \textit{real},

iv) \(\min\) is thus, because of the identification condition, a [prtr]-priority-SIX-operator.

If the first three conditions were not satisfied, then the search for an other defining occurrence would be continued in the same range, or failing that, in a surrounding range.)
4.4. Context conditions

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

4.4.1. The identification conditions

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

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

c) No proper program contains an indication which as an operator-applied occurrence identifies an operator-defining occurrence which as an indication-applied occurrence identifies an indication-defining occurrence different from the one identified by the given indication as an indication-applied occurrence.

(Condition c makes a program under certain circumstances improper independent of its elaboration. Without condition c, a program containing

\[
\begin{align*}
\text{priority} \times 2 \times \text{cp} & = \text{real a, b : skip;}
\text{random} < 0.5 \mid \text{priority} \times 2 \times 0.1 < 0.2)
\end{align*}
\]

would be improper if, during the elaboration of this clause, the value of random < 0.5 turns out to be true. Then, the presence of an indication-defining occurrence of ? in the serial-clause priority = 2; 0.1 < 0.2 causes its protection (6.4.2.a, 6.1.2.a, 6.0.2.d) to replace both occurrences of ? by another indication and thereby deprives the last occurrence of its operator-defining occurrence, which violates condition b. However, condition c makes the program improper immediately since the fourth occurrence of ? identifies the third as its indication-defining occurrence and the second as its operator-defining occurrence which itself identifies the first occurrence as its indication-defining occurrence.)

4.4.2. The uniqueness conditions

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

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

(e.g., none of the closed-clauses (6.4.1.a)
(real x; real y; sin (3.14)),
(real y; int y; sin (3.14)),
(real p; p -> p; sin (3.14)),
(mode a = real; mode a = real; sin (3.14)), or
(priority b = 8; priority b = 8; sin (3.14))
is contained in a proper program.)

c) No proper program contains a reach containing two operation-declarations
the operators of whose captions are the same terminal productions of a
notion ending with 'idication' and all of whose correspondent constituent
virtual-parameters (7.5.1.b, 7.1.1.x, 5.4.1.c, 7.1.1.y) are virtual-
declarers specifying modes loosely related to one another (4.4.3.c).

e.g., neither the closed-clause

\[
\text{op max} = (\text{int } a, \text{ int } b) \int : (a > b \mid a \mid b);
\]

\[
\text{op max} = (\text{int } a, \text{int } b) \text{ real } : (a > b \mid a \mid b); \sin (3.14)
\]

nor

\[
\text{op max} = (\text{int } a, \text{ref } \text{ int } b) \int : (a > b \mid a \mid b);
\]

\[
\text{op max} = (\text{ref } \text{ int } a, \text{int } b) \int : (a > b \mid a \mid b); \sin (3.14)
\]
is contained in any proper program, but

\[
\text{op max} = (\text{int } a, \text{ int } b) \text{ real } : (a > b \mid a \mid b);
\]

\[
\text{op max} = (\text{real } a, \text{ real } b) \text{ real } : (a > b \mid a \mid b); \sin (3.14)
\]

may be.)

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

4.4.3. The mode conditions

a) A mode \( M \) is "strongly coerced from" ("firmly coerced from") a mode \( N \) if
the notion 'N base' is a production of the notion 'strong M base' ('firm M base') (see 8.2); \( M \) is "united from" \( N \) if \( M \) is "union of LMODSETY \( N \)
RMODSETY mode" where "LMODSETY" ("RMODSETY") stands for any terminal pro-
duction of the metanotion 'LMODSETY' ('RMODSETY').

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

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

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

(e.g., the modes specified by proc real and ref real are related and,

hence, loosely related and those specified by \([\text{real}] \) and by \([\text{ref real}] \)
are loosely related but not related.)

d) No proper program contains a declarer (7.1.1.a) specifying a mode united
from (a) two modes related (b) to one another, nor does it contain two
declarers specifying modes united from two modes \( P \) and \( Q \) in which \( P \) and \( Q \)
are in a different order.
4.4.3. continued

(e.g., the declarer union (ref, real) is not contained in any proper program, and union (int, real) and union (real, int) may be, but then specify the same mode; see the remarks at the end of 7.1.1.)

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

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

4.4.4. The declaration condition

a) A mode-indication (4.2.1.b) contained in a declarer (7.1.1.a) is "shielded by that declarer if

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

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

iii) it is, or is contained in, a virtual-parameter (7.1.1.y), or

iv) it is, or is contained in, a virtual-declARSER following a virtual-parameters-pack (5.4.1.f), or

v) it is, or identifies, an indication-defining occurrence contained in that declARSER.

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

b) An actual-declARSER D (7.1.1.b) may "show" a mode-indication M (4.2.1.b); this is determined in the following steps:

Step 1: A copy is made of D; the copy is protected; each mode-indication is "shielded by that declARSER";

Step 2: If the copy is, or contains and does not shield (a), a mode-indication which is the same terminal production as M, then D shows M; otherwise, Step 3 is taken;

Step 3: If the copy is, or contains and does not shield, a not yet encountered mode-indication O, then O and all mode-indications consisting of the same sequence of symbols are said to have been encountered, O is replaced by a copy of the protected actual-declARSER of that mode-declARSER whose mode-indication is the indication-defining occurrence identified by O, and Step 2 is taken; otherwise, D does not show M.

(e.g., in the declaration

mode a = [1:2] b, b = union (ref d, ref real), d = struct (ref e e),

a = proc (int) a,

the mode-indications shown by [1:2] b are b and d.)

c) No proper program contains a mode-declaration (7.2.1.a) whose mode-indication is shown by its actual-declARSER.

(e.g., none of the declarations

mode a = a,

mode b = e, e = [1:10] b,

mode d = [1:2] ref union (proc (d) d, proc d),

mode person = struct (int age, person uncle)

is contained in any proper program.)
5. Denotations

(Denotations, e.g., 3.1d or "abc", are terminal productions of notions whose value is independent of the elaboration of the program. In other languages, they are sometimes termed "literals" or "constants").

5.0.1. Syntax

a) denotation : PLAIN denotation(510b, 511a, 512a, 513a, 514a) ;
   BITS denotation(52b) ; row of character denotation(53b) ;
   procedure with PARAMETERS MOLD denotation(54b) ;
   format denotation(55a).

   {Examples:
       a) 3.1d ; i 0 i ; "algol_report" ; ((bool a) int : (a | i | 0)) ; $5d$}

5.0.2. Semantics

   Each denotation possesses a new instance of one same value whose mode
   is that enveloped {1.1.6.j} by its original {1.1.6.c}; its elaboration in-
   volves no action.

   {e.g., the value of "algol_report" which is a production of 'row of
   character denotation' is of the mode 'row of character'.}

5.1. Plain denotations

   (Plain-denotations are those of arithmetic values, truth values and
   characters, e.g., i, 3.1d, true and "a").

5.1.0.1. Syntax

a) plain denotation : PLAIN denotation(510b, 511a, 512a, 513a, 514a).

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

   {Examples:
       b) long 0 ; long_long 3.1415926535 8979323846 2843383279 5028841971 69399}

5.1.0.2. Semantics

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

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

5.1.1.1. Syntax

a) integral denotation(510b,512e,d,f,i,55g,860a) : digit token(303c) sequence.

(Examples:
  a) \(0; 4086; 00123\) (Note that \(-1\) is not an integral-denotation.)
)

5.1.1.2. Semantics

The a priori value of an integral-denotation is the integer which in decimal notation is that integral-denotation in the representation language (1.1.6). (See also 5.1.0.2.b.)

5.1.2. Real denotations

5.1.2.1. Syntax

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

b) variable point numeral(a,f) : integral part(c) option, fractional part(d).

c) integral part(b) : integral denotation(511a).

d) fractional part(b) : point symbol(31b), integral denotation(511a).

e) floating point numeral(a) : stagnant part(f), exponent part(g).

f) stagnant part(e) : integral denotation(511a) ; variable point numeral(b).

g) exponent part(e) : times ten to the power choice(h), power of ten(i).

h) times ten to the power choice(g) :
   times ten to the power symbol(31b) ; letter e(302b).

i) power of ten(g) : plus minus(304c) option, integral denotation(511a).

(Examples:
  a) \(0.000123 ; 1.23\times 10^{-4} \)
  b) \(.123 ; 0.123 \)
  c) \(123 \)
  d) \(.123 \)
  e) \(1.23\times 10^{-4} \)
  f) \(1 ; 1.23 \)
  g) \(e^{-4} \)
  h) \(10 ; e \)
  i) \(3 ; 4.6 ; -678 \)
)

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 10 as many times as there are digit-tokens in the fractional-part.

b) The a priori value of a variable-point-numeral is the sum in the sense of numerical analysis of 0, the a priori value of its integral-part, if any, and that of its fractional-part (See also 5.1.0.2.b).

c) The a priori value of an exponent-part is 10 raised to the a priori value of the integral-denotation of its power-of-ten if that power-of-ten does not begin with a minus-symbol; otherwise, it is \(1/10\) raised to the a priori value of that integral-denotation.

d) The a priori value of a floating-point-numeral is the product in the sense of numerical analysis of the a priori values of its stagnant-part and exponent-part (See also 5.1.0.2.b).
5.1.3. Boolean denotations

5.1.3.1. Syntax

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

{Examples:
  a) true ; false }

5.1.3.2. Semantics

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

5.1.4. Character denotations

{Character-denotations consist of a string-item between two quote-
symbols, e.g., "a". To indicate a quote, a double quote-symbol is used for
the string-item: "". Since the syntax nowhere allows character- or string-
denotations to follow one another, ambiguities do not arise.}

5.1.4.1. Syntax

a) character denotation(55j,k,860a):
   quote symbol(31i), string item(b), quote symbol(31i).

b) string item(a,53b):
   character token(309a) ; quote image(c) ; other string item(1.1.5.c).

c) quote image(b) : quote symbol(31i), quote symbol(31i).

{Examples:
  a) "a"
  b) a ; "" ; ?
  c) ""

5.1.4.2. Semantics

a) Each string-item possesses a unique character. {The character possessed
   by a quote-image (space-symbol, digit-zero, digit-token, point-symbol,
   times-ten-to-the-power-choice, plus-i-times-symbol, plus-symbol) may be
   termed a quote (space, zero, digit, point, times ten to the power, plus i
   times, plus).}

b) The value of a character-denotation is a new instance of the character
   possessed by its string-item.

5.2. Bits denotations

{There are two kinds of denotations of structured or multiple values,
  viz., bits-denotations, e.g., 1 0 1 1, and string-denotations, e.g., "abc".
  These denotations differ in that a string-denotation contains zero or two or
  more string-items but a bits-denotation may contain one or more flipflops.
  (See also character-denotations 5.1.4.)}
5.2.1. Syntax

a) bits denotation := BITS denotation(b,c).
b) structured with row of boolean field LENGTH LENGTHY letter aleph
denotation(b,860a) : long symbol(31d), structured with row of boolean
field LENGTHY letter aleph denotation(b,c).
c) structured with row of boolean field letter aleph denotation(b,860a):
flipflop(303e) sequence.

{Examples:
  b) Long 1 0 1 1;
  c) 1 0 1 1}

5.2.2. Semantics

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

5.3. String denotations

5.3.1. Syntax

a) string denotation : row of character denotation(b).
b) row of character denotation(551,k,860a) : quote symbol(311),
  string item(514b) sequence proper option, quote symbol(311).

{Examples:
  b) "a" ; "aba" ; """a+b"" ; is_a_formula"

5.3.2. Semantics

The value of a string-denotation is a multiple value (2.2.3.3) whose de-
scriptor consists of an offset \( l \) and one quintuple \([l,n,l,1,1]\), where \( n \) stands for the number of string-items contained in the string-denotation;
for \( i = l, \ldots, n \), the element with index \( i \) of that multiple value is a new
instance of the character possessed by the \( i \)-th constituent string-item of
the string-denotation.

(The construction "a" is a character-denotation, not a string-denotation.
However, in all strong positions, e.g., string \( s := "a" \), it can be rowed to
a multiple value (8.2.6). Elsewhere, where a multiple value is required, a
cast (8.3.4.1.a) may be used, e.g., union (int, string) is := string : "a".
The "string", i.e., value of mode 'row of character', possessed by
"""a+b"" is_a_formula" may well be presented informally as follows:
"a + b" is a formula.)
5.4. Routine denotations

(A routine-denotation, e.g., ((real a, b) real : (a > b | b | a)), always has a formal-parameters-pack, e.g., (real a, b). To the right of this formal-parameters-pack stands a cast, e.g., real : (a > b | b | a), whose virtual-declarer specifies the mode of the value, if any, delivered by the elaboration of the routine, e.g., real. The whole is enclosed between an open-symbol and a close-symbol, but these may often be omitted, see the extension 9.2.4. It is essential that, in general, a routine-denotation be closed, for, otherwise, denotations like (int sintsoff) : (int branquart) : bool (wodon) could also be calls, or formulas like (int a) int : 1 + 2 + 3 would be ambiguous if + is also declared as an operator accepting a routine as left operand.)

5.4.1. Syntax

a)* routine denotation : procedure with PARAMETERS MOID denotation(b).
   b) procedure with PARAMETERS MOID denotation(860a) : open symbol(31e),
      formal PARAMETERS(c,e) pack, MOID cast(834a), close symbol(31e).
   c) VICTAL PARAMETERS and PARAMETER(b,c,71x,862a) :
      VICTAL PARAMETERS(c,e,71y,74b), gamma{a}, VICTAL PARAMETER(e,71y,74b).
   d) gamma(a) : go on symbol(31f) ; comma symbol(31e).
   e) formal MODE parameter(b,c,74a) :
      formal MODE declarer(71b), MODE mode identifier(41b).
   f)* VICTAL parameters pack : VICTAL PARAMETERS(c,e,71y,74b) pack.

   (Examples:
   b) (bool a, b) bool : (a | b | false)) ;
   c) 1 : real a; [1 : | a] real b ;
   d) ; ; ;
   e) bool a )

5.4.2. Semantics

A routine-denotation possesses that routine which can be obtained from it in the following steps:
Step 1: A copy is made of the routine-denotation;
Step 2: An equals-symbol followed by a skip-symbol is inserted in the copy following the last identifier in each copied constituent formal-parameter of the formal-parameters-pack of the routine-denotation; the open-symbol of that formal-parameters-pack is deleted and its close-symbol is replaced by a go-on-symbol;
Step 3: If the cast of the routine-denotation is a void-cast, then an open-symbol is inserted in the copy preceding, and a close-symbol following that cast; the copy, thus modified, is the routine possessed by the routine-denotation.

(The routine possessed by p1 after the elaboration of the identity-declaration (7.4.1.a) proc p1 = (int a, b) real : (a > b | xx | yy) is (int a = , int b = , real : (a > b | xx | yy)) and that possessed by p2 after the elaboration of proc p2 = (real a; real b) : (a > b | stop) is (real a = , real b = , (a > b | stop)). A routine is the same sequence of symbols as some closed-clause (6.3.1.a). For the use of routines, see 8.4 (formulas), 8.2.2 (deproocedured-coercveis) and 8.6.2 (calls).}
5.5. Format denotations

5.5.1. Syntax

a) format denotation(860a) :
   formatter symbol(31b), collection(b) list, formatter symbol(31b).
b) collection(a,b) : picture(c) ; insertion(d) option, replicator(f),
   collection(b) list pack, insertion(a) option.
c) picture(b) : MODE pattern(552a,553a,554a,555a,556b,557b,-) option,
   insertion(a) option.
d) insertion(b,c,m,552b,f,554a,557b) :
   literal(j) option, insert{e} sequence ; literal(j).
e) insert(d) : replicator(f), alignment(i), literal(j) option.
f) replicator(b,e,j,n) : replication(g) option.
g) replication(f,k,557b) :
   dynamic replication(h) ; integral denotation(511a).
h) dynamic replication(g) :
   letter n(302b), strong CLOSED integral clause(63a,640a,-).
i) alignment(e) : letter k(302b) ; letter x(302b) ;
   letter y(302b) ;
   letter l(302b) ;
   letter p(302b).
j) literal(e,d,552r,554b) : replicator(f), STRING denotation(514a,53b),
   replicated literal(k) sequence option.
k) replicated literal(j) : replication(g), STRING denotation(514a,53b).

{Examples:}
a) $p$"table_of"z10a,n(lim-1)({x+y}+{x+y}+{x+y}+{x+y}+{x+y}+{x+y}+{x+y}+{x+y}+{x+y})p$ ;
b) p"table_of"z10a ; 3x3(2x+12x+2d)+x"&k"10x+10d+2d+1)p$ ;
c) L50ka("mon","tue","wednes","thurs","fri","satur","sun")"day" ; p ;
d) p"table_of"x ; "day" ;
e) p"table_of"m ;
f) n(lim-1) ; 10 ;
g) n(lim-1) ;
j) "&x" ;
k) 20"_" ;

l) sign mould(552a,553a,d,e) :
   loose replicatable zero frame(m), sign frame(p) ; loose sign frame(m).
m) loose ANY frame(1,552a,553b,d,555a,556b,557b) :
   insertion(a) option, ANY frame(n,p,q,557c).

n) replicatable ANY frame(s) : replicator(f), ANY frame(o,q).
o) zero frame(n,552e) : letter z(302b),
p) sign frame(1,m) : plusminus(304c).
q) suppressible ANY frame(m,n,557b) :
   letter s(302b) option, ANY frame(552e,553a,f,555b,556c).

{Examples:}
l) "="18a+ ; 2a+ ;
m) "="18a ;
n) 12a ;
q) "&t" ; 10a }

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

bb) A format may be associated with a file by a call of `format (10.5.3.a), `outf (10.5.3.1.a) or `inf (10.5.3.2.a), which causes a transform to be elaborated (5.5.8.1.a), the collection-list of the format-denotation considered in 5.5.8.2.b. Step 2 to be unfolded (cc), the result to be the current picture-list of the file and its first constituent picture to be the current picture of the file; e.g., after the call `format (f1, $pt, $i{3d.d}$l$)#), the current picture-list of the file f1 is pt, 3d.d, 3d.d, 3d.dl and the current picture is pt).

c) The result of unfolding a collection-list (10.5.3.b) is a picture-list obtained as follows:
a) if the collection-list is a picture, then the result consists of that picture;
b) if the collection-list is a collection but not a picture, then the result consists of the first insertion-option of the collection, followed by as many copies of the result of unfolding the collection-list of its collection-list-pack as is the value of its replicator, separated by comma-symbols, followed by its last insertion-option; e.g., the result of unfolding `3k"ab"2(10a)l is 3k"ab"10a, 10al);
c) if the collection-list is a collection-list-proper, then the result consists of the result of unfolding the collection of that collection-list-proper followed by a comma-symbol, followed by the result of unfolding its collection-list; e.g., the result of unfolding `10a, p{n(t)$(d.2d)" is 10a, p" when the value of t is 0).

dd) When one of the formatted-tranput routines `outf (10.5.3.1.a), `out (10.5.3.1.b), `inf (10.5.3.2.a) or `in (10.5.3.2.b) is called, then tranput takes place in the following steps:
Step 1: The values to be tranput are elaborated collaterally and the result is "straightened" (10.5.0.2) into a series of values, the first of which, if any, is made to be the current value;
Step 2: If the current picture of the file is an insertion-option, then its insertion, if any, is performed (ee), the next picture, if any, is made to be the current picture of the file and Step 2 is taken; otherwise, Step 3 is taken;
Step 3: If the series of values is empty or exhausted, then the tranput is accomplished; otherwise, if the picture-list is exhausted, then `format and of the file is called, a routine which may be provided by the programmer (10.5.1.kk);
Step 4: If the current value is "compatible" with (nn) the current picture, then that value is tranput under control of that picture; otherwise, value error of the file is called, a routine which may be provided by the programmer;
Step 5: The next value, if any, is made to be the current value, the next picture, if any, is made to be the current picture and Step 2 is taken.

ee) The value of the empty replicator is 1; the value of a replication which is an integral-denotation is the value of that denotation; the value of a dynamic-replication is the value of its integral-clause if that value is positive, and 0 otherwise.

ff) Tranput occurs at the current "position" (i.e., page number, line number and char number) of the file. At each position of the file within certain limits (10.5.1.1.k,l,m) some character is "present", depending on the contents of the file and on its "conversion key" (10.5.1.11).
5.5.1. continued 2

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

hh) Performing an alignment affects the position of the file as follows, where \( n \) stands for the value of the preceding replicator:
   a) letter-\( k \) causes the current char number to be set to \( n \);
   b) letter-x causes the char number to be incremented by \( n \) (10.5.1.2.o);
   c) letter-y causes the char number to be decremented by \( n \) (10.5.1.2.p);
   d) letter-l causes the line number to be incremented by \( n \) and the char number to be reset to \( l \) (10.5.1.2.q);
   e) letter-p causes the page number to be incremented by \( n \) and both the line number and the char number to be reset to \( l \) (10.5.1.2.r).

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

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

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

ll) On output, a picture may be used to "edit" a value in the following steps:
Step 1: The value is converted by an appropriate output routine (10.5.2.1.c, d,e) to a string of as many characters as specified by the picture; if the pattern of the picture is an integral-pattern, then this conversion takes place to a base equal to the value of the integral-denotation which is the same sequence of symbols as its constituent radix, if any, and base 10 otherwise; if this number of characters is not sufficient, then value error of the file is called, a routine which may be provided by the programmer (10.5.1.kk); Step 2: In those parts, if any, of the string specified by a sign-where, a character specified by the sign-frame will be used to indicate the sign, viz., if the sign-frame is a minus-symbol and the value is nonnegative,
then a space, and, otherwise, the character specified by the sign-frame; this character is shifted in that part of the string specified by the sign-mould as far to the right as possible across all leading zeroes, and those zeroes are replaced by spaces (; e.g., under the sign-mould 4st, the string possessed by "+0005" becomes that possessed by "...+5"); if the picture does not contain a sign-mould and the value is negative, then value error of the file is called;

Step 3: Leading zeroes in those parts of the string specified by any remaining zero-frames are replaced by spaces (; e.g., under the picture 62d0a, the integer 180168 becomes the string possessed by "18.168");

Step 4: For all frames occurring in the picture, first the preceding insertion, if any, is performed, and next, if the frame is not "suppressed" (i.e., preceded by letter-s), then that part of the string specified by the frame is written; finally, the insertion, if any, following the last constituent frame is performed (; e.g., editing under the picture 62d=2d=2d=, the integer 180168 causes the string possessed by "18.1-1.688" to be written).

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

Step 1: A string is obtained consisting of the characters obtained by performing the following process for all frames occurring in the picture, viz., first, the insertion, if any, preceding the frame is performed and next, as many characters are obtained as are specified by the frame; each of those characters is obtained,

if the frame is not suppressed, then by reading from the file a character,
and, if the frame is a digit- (point-, exponent-, complex-)frame and the character is not a digit (point, times ten to the power, plus i times), then calling char error of the file (10.5.1.kk) with as its parameter a zero (point, times ten to the power, plus i times), and
if the frame is suppressed, then by taking, if the frame is a digit-
(zero-, point-, exponent-, complex-, character-)frame a zero (zero,
point, times ten to the power, plus i times, space);

Step 2: Those parts, if any, of the string specified by a sign-mould must contain a character, specified by the sign-frame, to indicate the sign (; see 11. Step 2); if those parts contain such a character, with only spaces appearing in front of it and no leading zeroes appearing after it, then those leading spaces, if any, are deleted; otherwise, char error is called with a plus; if this character is a space, and the sign-frame is a minus-symbol, then it is replaced by a plus (; e.g., if in Step 1 under control of $x-a$, the string possessed by "...3" is obtained, then in Step 2 that possessed by "+3" is obtained);

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

Step 4: The string is converted by an appropriate input routine (10.5.2.2.c, d,e) into a value of the given mode, if possible, and, otherwise, value error of the file is called (; e.g., if the value of maxint (10.1.b) is 10000, then under +6d it is possible to input +10000, but not +10001).

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

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

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

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

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

pp) If a value is transput under control of a picture whose constituent
pattern is not an integral-choice-pattern (5.5.2.f), boolean-pattern
(5.5.4.a) or string-pattern (5.5.7.b), then on output (input) it is edited
(indited) under control of the picture.

5.5.2. Syntax of integral patterns

a) integral pattern(55c) : radix mould(b) option, sign mould(55l) option,
integral mould(d) ; integral choice pattern(f).
b) radix mould(a) : insertion(55a) option, radix(c), letter r(302b).
c) radix(b) : digit two(303d) ; digit four(303d) ; digit eight(303d) ;
digit one(303d) ; digit zero(303d) ; digit one(303d) ; digit six(303d).
d) integral mould(a,553b,d,e) :
loose replicatable suppressible digit frame(55m) sequence.
e) digit frame(55g) : zero frame(55o) ; letter d(302b).
f) integral choice pattern(a) :
insertion(55d) option, letter c(302b), literal(55j) list pack.

{Examples:
 a) 2r4d30ad ; 12s+d ; sd"-sd"-19"2d ;
 l30ka("mon","tues","wednes","thura","fri","satur","sun") ;
b) 2r ;
c) 2 ; 4 ; 8 ; 10 ; 16 ;
d) sd"-sd"-19"2d ;
f) l30ka("mon","tues","wednes","thura","fri","satur","sun")
}

{If a given value is transput under control of a picture whose constitu-
ent pattern is an integral-choice-pattern, then the insertion, if any, pre-
ceding the letter-c is performed, and,
a) on output, letting n stand for the integer to be output, if n > 0 and
the number of literals in the constituent literal-list-pack is at least
n, then the n-th literal is written on the file; otherwise, the further
elaboration is undefined;
b) on input, one of the constituent literals of the constituent literal-
list-pack is expected on the file; if the i-th constituent literal is
the first one present, then the value is i; if none of these literals is
present, then the further elaboration is undefined;
c) finally, the insertion, if any, following the pattern is performed.}
5.5.3. Syntax of real patterns

a) real pattern(55c, 556b) : sign mould(55l) option, real mould(b) ;
   floating point mould(a).
b) real mould(a, e) : integral mould(552a),
   loose suppressible point frame(55m), integral mould(552d) option ;
   loose suppressible point frame(55m), integral mould(552d).
c) point frame(55q) : point symbol(31b).
d) floating point mould(a) : stagnant mould(e), loose suppressible exponent
   frame(55m), sign mould(55l) option, integral mould(552d).
e) stagnant mould(d) : sign mould(55l) option, INTEGER mould(b, 552d, -).
f) exponent frame(55q) : letter e(302b).

(Examples:
  a) +.12d ; +d.11de+2d ;
  b) d.11d ; .12d ;
  d) +d.11de+2d ;
  e) +d.11d )

5.5.4. Syntax of boolean patterns

a) boolean pattern(55c) : insertion(55d) option, letter b(302b),
   boolean choice mould(b) option.
b) boolean choice mould(a) : open symbol(31e), literal(55j),
   comma symbol(31e), literal(55j), close symbol(31e).

(Examples:
  a) l"result"12xb ; 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
("", "") ). If a given value is transput under control of a picture whose
constituent pattern is a boolean-pattern, then the insertion, if any, pre-
ceding the letter-b is performed, and,
a) on output, if the truth value to be output is true, then the first con-
stituent literal of the constituent choice-mould is written, and, other-
wise, the second;
b) on input, one of the constituent literals of the constituent choice-
mould is expected 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 unde-
fined;
c) finally, the insertion, if any, following the pattern is performed.)

5.5.5. Syntax of character patterns

a) character pattern(55c) : loose suppressible character frame(55m).
b) character frame(55q) : letter a(302b).

(Example:
  a) ".a" )
5.5.6. Syntax of complex patterns

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

   (Example:
b) 2e+.12de+2d"a"i "ei+.10de+2d }

5.5.7. Syntax of string patterns

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

   (Examples:
b) lt ; 5a5a5a ; p"table_of"z10a (Note that a is a character-pattern,
   whereas la is a string pattern for a string with one element.)

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

5.5.8. Transformats

(Transformats are exclusively used as actual-parameters of formatted-
transput routines; for reasons of efficiency, the programmer has deliber-
ately been made unable to use them elsewhere by the choice of the field-
selector, which contains letter-aleph for which no representation is pro-
vided. Although transforms are not denotations at all, they are handled 
here because of their close connection to formats.)

5.5.8.1. Syntax

a) structured with row of character field letter aleph digit one
   transformat(74b) : firm format unit(61e).

   (Example:
a) (x ≥ 0 | $5d$ | $5d"-\$) }
5.5.8.2. 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 transfomrat is elaborated in the following steps:
Step 1: It is preelaborated \( 1.1.6.i \);
Step 2: A format-denotation is considered which is the same sequence of symbols as the format obtained in Step 1;
Step 3: All constituent dynamic-replications \( 5.5.1.h \) of the considered format-denotation are elaborated collaterally \( 6.2.2.a \), where the elaboration of a dynamic-replication is that of its integral-clause;
Step 4: Each of those dynamic-replications is replaced by an integral-denotation \( 5.1.1.1.a \) which possesses the same value as that dynamic-replication if that value is positive, and, otherwise, by a digit-zero; furthermore, every replicator which is empty is replaced by a digit-one;
Step 5: That string-denotation \( 5.3.1.a \) (character-denotation \( 5.1.4.1.a \)) is considered which is obtained by replacing in the considered format-denotation as modified in Step 4 each constituent quote-symbol by a quote-image \( 5.1.4.1.c \) and the first and the last constituent formatter-symbol by a quote-symbol;
Step 6: A new instance of the value of the considered string-denotation (of a multiple value composed of the value of the considered character-denotation as its {only} element and of a descriptor consisting of an offset \( 1 \) and one quintuple \( \{1,1,1,1,1\} \)) is made to be the (only) field of a new instance of a structured value \( 2.2.3.2 \) whose mode is that enveloped \( 1.1.6.3.j \) by the original \( 1.1.6.c \) of the transfomrat;
Step 7: The transfomrat is made to possess the structured value obtained in Step 6.
6. Phrases

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

6.0.1. Syntax

\begin{itemize}
\item[a]* SOME phrase : \\
\texttt{SORTITY SOME PHRASE(61a,62a,b,c,d,f,63a,64a,c,d,e,70a,81a,-)}.
\item[b]* SOME expression : \\
\texttt{SORTITY SOME MODE clause(61a,62b,c,d,f,63a,64a,c,d,e,81a)}.
\item[c]* SOME statement : strong SOME void clause(61a,62b,63a,64a,c,d,e,81a).
\end{itemize}

(The rules b and c are not actually used in this Report but serve to help the reader, who may know some such constructions in other languages under those appellations. For an informal introduction into ALGOL 68 (6.1.1) also the following rules may be helpful:

\begin{itemize}
\item[d]* MODE constant : \texttt{MODE FORM(830a,844g,850a,860a)}.
\item[e]* MODE variable : \texttt{reference to MODE FORM(830a,844g,850a,860a)}.
\item[f]* procedure : \texttt{REPETY PROCEDURE FORM(830a,844g,850a,860a)}.
\item[g]* structure display :
\begin{itemize}
\item strong collateral structured with \texttt{FIELDS} and \texttt{FIELD} clause(62f).
\end{itemize}
\item[h]* row display : \texttt{SORTITY collateral row of MODE clause(62c,d,-)}.
\end{itemize}

6.0.2. Semantics

\begin{itemize}
\item[a] The elaboration of a phrase begins when it is initiated, it may be "interrupted", "halted" or "resumed", and it ends by being "terminated" or "completed", whereupon, if the phrase "appoints" a unitary-phrase as its "successor", then the elaboration of that unitary-phrase is initiated.
\item[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 (2.3.b) do not permit satisfactory elaboration. (Whether, after an interruption, the elaboration of the phrase is resumed, the elaboration of some unitary-phrase is initiated or the elaboration of the program ends, is left undefined in this Report.)
\item[c] The elaboration of a phrase may be halted (10.4.c), i.e., no further actions constituting the elaboration of that phrase take place until the elaboration of the phrase is resumed (10.4.d), if at all.
\end{itemize}
6.0.2. continued

4) A given (serial-)clause is "protected" in the following steps:
Step 1: If the given clause contains a defining occurrence (4.1.2.a) (an indication-defining occurrence (4.2.2.a)) of a terminal production of a notion ending with 'identifier' ('indication') which also occurs outside it, then that defining (indication-defining) occurrence and all occurrences identifying it are replaced by occurrences of one same terminal production of that notion which does not occur in the program and Step 1 is taken; otherwise, Step 2 is taken;
Step 2: If the given clause as possibly modified in Step 1, 2 or 4 contains an operator-defining occurrence (4.3.2.a) of a terminal production of a notion ending with 'indication' which also occurs outside it, then that operator-defining occurrence and all occurrences identifying it are replaced by occurrences of one same new terminal production of that notion which does not occur in the program and Step 3 is taken; otherwise, the protection of the given clause is accomplished;
Step 3: If the indication is a dyadic-indication, then Step 4 is taken; otherwise, Step 2 is taken;
Step 4: A copy is made of the priority-declaration containing the indication which, before the replacement in Step 2, was identified by that operator-defining occurrence; that indication in the copy is replaced by an occurrence of the new terminal production; the given clause is modified by inserting before it the thus modified copy of the priority-declaration followed by a go-on-symbol, and Step 2 is taken.

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

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

6.1. Serial clauses

(Serial-clauses are built from unitary-clauses and declarations with the help of go-on-symbols (;), completion-symbols (., or exit) and labels, e.g.,
\( x > 0 \mid x := i \mid i \); \( y.\ l: \ y + i \), where the value of the clause is that of \( y \), if \( x > 0 \) and that of \( y + 1 \) otherwise. A serial-clause may begin with declaration-preludes, e.g., \( int\ n := i \); \( int\ n := i \); \( x := y + n \). Labels may occur in only three syntactic positions within serial-clauses: after a completion-symbol (here a label is obligatory, e.g., \( .l: \)), in a sequencer (e.g., \( .l: \)), or at the beginning of a clause-train (i.e., one or more unitary-clauses separated by sequencers, e.g., \( l: x := i; y := i \)).
A declaration-prelude may begin with void-clauses (statements), e.g., in order to supply a multiple value as in \([i:n]\ real\ x1;\ for\ i\ to\ n\ do\ x1\ [i] := t \times i;\ real\ y; \); however, these void-clauses may not be labelled. A declaration-prelude always ends with a go-on-symbol. The modes of some serial-clauses must be balanced (6.1.1.g). For remarks concerning the balancing of modes see 6.4.1.)
6.1.1. Syntax

a) SORTKEY serial CLAUSE(63a,64b,e) : declaration prelude(b) sequence
    option, suite of SORTKEY CLAUSE trains(f,g).
b) declaration prelude(a,2b, c):
    statement prelude(c) option, single declaration(d), go on symbol(31f).
c) statement prelude(b) : chain of strong void units(e) separated by
    go on symbols(31f), go on symbol(31f).
d) single declaration(b) :
    unitary declaration(70a) ; collateral declaration(62a).
e) SORTKEY MOID unit(c,1,558a,62b,c,e,h,74b,831c,834a) :
    SORTKEY unitary MOID clause(81a).
f) suite of SORTKEY CLAUSE trains(a,g) :
    chain of SORTKEY CLAUSE trains(h) separated by completers(l).
g) suite of FEAT CLAUSE trains(a,g) : FEAT CLAUSE train(h) ;
    FEAT CLAUSE train(h), completer(l), suite of strong CLAUSE trains(f) ;
    strong train(h), completer(l), suite of FEAT CLAUSE trains(g).
h) SORTKEY MOID clause train(f,g,2g) : label(k) sequence option,
    statement interlude(i) option, SORTKEY MOID unit(e).
i) statement interlude(h,2f) :
    chain of strong void units(e) separated by sequencers(j), sequencer(j).
j) sequencer(1,30c) : go on symbol(31f), label(k) sequence option.
k) label(h,j,i,2d) : label identifier(41b), label symbol(31e).
l) completer(f,g,30c) : completion symbol(31f), label(k).

{Examples:}
a) \begin{verbatim}
real a := 0; \ L1: L2: x := a + 1; \ (p \ L3) ; \ (x > 0 \ L3 \ x := 1 - x); false.
\end{verbatim}
b) \begin{verbatim}
real a := 0; ; read (n); [1:m] real x1, y1 ;
\end{verbatim}
c) \begin{verbatim}
real (n); ;
\end{verbatim}
d) \begin{verbatim}
real a := 0 ; [1:m] real x1, y1 ;
false ;
\end{verbatim}
f) \begin{verbatim}
L1: L2: x := a + 1; \ (p \ L3) ; \ (x > 0 \ L3 \ x := 1 - x); false.
\end{verbatim}
h) \begin{verbatim}
L1: L2: x := a + 1; \ (p \ L3) ; \ (x > 0 \ L3 \ x := 1 - x); false ;
\end{verbatim}
i) \begin{verbatim}
x := a + 1; \ (p \ L3) ; \ (x > 0 \ L3 \ x := 1 - x); ;
\end{verbatim}
j) ; ; l4: l5: ;
k) l4: ;
l) \ L3: }

6.1.2. Semantics

a) The elaboration of a serial-clause is initiated by protecting (6.0.2.a)
    it and then initiating the elaboration of its textually first constituent
    single-declaration or unitary-clause.

b) The completion of the elaboration of a single-declaration or unitary-
    clause preceding a go-on-symbol followed (not followed) by a label-sequence
    initiates the elaboration of the unitary-clause following that label-
    sequence (the single-declaration or unitary-clause following that go-on-
    symbol).

c) The elaboration of a serial-clause is
    interrupted (halted, resumed) upon the interruption (halting, resump-
    tion) of the elaboration of a constituent single-declaration or unitary-
    clause;
    terminated upon the termination of the elaboration of a constituent
    single-declaration or unitary-clause appointing a successor outside the
    serial-clause, and that successor (8.2.7.2.b.Step 1) is appointed the suc-
    cessor of the serial-clause.
6.1.2. continued

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

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

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

6.2 Collateral phrases

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

6.2.1. Syntax

a) collateral declaration(61d) : unitary declaration(70a) list proper.
b) strong collateral void clause(2a,81d) :
   parallel symbol(31e) option, strong void unit(61e) list proper PACK.
c) STRONGENCY collateral row of MODE clause(81d) :
   STRONGENCY MODE unit(61e) list proper PACK.
d) firm collateral row of MODE clause(81d) : firm MODE balance(e) PACK.
e) firm MODE balance(d,e) :
   firm MODE unit(61e), comma symbol(31e), strong MODE unit(61e) list ;
   strong MODE unit(61e), comma symbol(31e), firm MODE unit(61e) ;
   strong MODE unit(61e), comma symbol(31e), firm MODE balance(e).
f) strong collateral structured with FIELDS and FIELD clause(81d) :
   strong structured with FIELDS and FIELD structure(g) PACK.
g) strong structured with FIELDS and FIELD structure(f,g) :
   strong structured with FIELDS structure(g,h), comma symbol(31e),
   strong structured with FIELD structure(h).
h) strong structured with MODE field TAG structure(g) :
   strong MODE unit(61e).
6.2.1. continued

[Examples:
  a) \texttt{real } \texttt{x}, \texttt{real } \texttt{y} ; \texttt{ (and by 9.2.c) real } \texttt{x}, \texttt{y} ;
  b) \texttt{(x := 1, y := 2, z := 3)} ;
  c) \texttt{(x, n)} ;
  d) \texttt{(1.2, 3, 4)} \texttt{ (in (1.2, 3, 4) + x1, supposing + has been declared also}
      \texttt{for 'row of real') ;}
  e) \texttt{1.2, 3, 4} \texttt{ (in (1.2, 3, 4) + x1) ; 1, 2.3} \texttt{ (in (1, 2.3) + x1)} ;
      \texttt{l, 2.3, 4} \texttt{ (in (l, 2.3, 4) + x1)} ;
  f) \texttt{(1, 2.3)} \texttt{ (in z := (1, 2.3)) ;}
  g) \texttt{1, 2.3} ;
  h) \texttt{l} ]

6.2.2. Semantics

a) If constituents of an occurrence of a terminal production of a notion
   are "elaborated collaterally", then this elaboration is the collateral
   action (2.2.5) consisting of the {merged} elaborations of these constituents,
   and is
   initiated by initiating the elaboration of each of these constituents,
   interrupted upon the interruption of the elaboration of any of these
   constituents,
   completed upon the completion of the elaboration of all of these con-
   stituents, and
   terminated upon the termination of the elaboration of any of these con-
   stituents, and if that constituent appoints a successor, then this is the
   successor of the occurrence.

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

c) A collateral-clause is elaborated in the following steps:
   Step 1: Its constituent units are elaborated collaterally (a);
   Step 2: If the terminal production of the metanotion 'MOD' enveloped
           (1.1.6.c) by the original (1.1.6.c) of the collateral-clause is a mode,
           then this mode is considered and Step 3 is taken; otherwise, (it is 'void'
           and) the elaboration of the collateral-clause is complete;
   Step 3: If one of the values of the units obtained in Step 1 is a name
           (2.2.3.5) which refers to a component of a multiple value having one or
           more states (2.2.3.3) equal to 0, then the further elaboration is undefin-
           ed; otherwise, Step 4 is taken;
   Step 4: If the considered mode begins with 'row of', then Step 5 is taken;
           otherwise, new instances of the values obtained in Step 1 are made, in the
           given order, to be the fields of a new instance of a structured value
           (2.2.3.2); this structured value is considered and Step 7 is taken;
   Step 5: Let \( m \) stand for the number of constituent units in the collateral-
           clause; if the considered mode begins with 'row of row of', then Step 6 is
           taken; otherwise, a new instance of a multiple value is created whose ele-
           ment with index \( i \) is a new instance of the value of the \( i \)-th constituent
           unit, and whose descriptor consists of an offset \( l \) and one quintuple
           \( [l,m,i,l,1] \); this multiple value is considered and Step 7 is taken;
6.2.2. continued

Step 6: If not all corresponding upper (lower) bounds of the multiple values obtained in Step 1 are equal, then the further elaboration is undefined; otherwise, the elements with indices \( \{i - 1\} \times n + j \), \( j = 1, \ldots, n \), of the new value, where \( n \) stands for the number of elements in one of those values, are new instances of the elements, taken in the order of ascending indices, of the value of the \( i \)-th constituent unit and the descriptor of the new value is a copy of the descriptor of the value of one of the constituent units into which an additional quintuple \( \{l, m, 1, 1, 1\} \) has been inserted before the old first quintuple, the offset has been set to 1, \( d_n \) has been set to 1, and for \( i = n, n - 1, \ldots, 2 \), the stride \( d_i - 1 \) has been set to \( (u_i - l_i + 1) \times d_i \); this new multiple value is considered and Step 7 is taken;

Step 7: The value of the collateral-clause is the considered value; its mode is the considered mode.

6.3. Closed clauses

(Closed-clauses are generally used to construct primaries (6.1.4.a) from serial-clauses, e.g., \( (x + y) \) in \( (x + y) \times a \). The question of identifica-
tion (Chapter 4) and protection (6.2.2.d) may arise in closed-clauses, be-
cause a serial-clause is a range (4.1.1.e) and it may begin with a declaration- prelude (6.1.1.a).)

6.3.1. Syntax

a) \textbf{SORTIFY closed CLAUSE}(2d,55h,81a) : \textbf{SORTIFY serial CLAUSE}(61a) \textbf{PACK}.

(Examples:

\begin{verbatim}
begin \texttt{i := i + 1; j := j + 1 end \{ (x + y) \}}
\end{verbatim}

6.3.2. Semantics

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

6.4. Conditional clauses

(Conditional-clauses allow the programmer to choose one out of a pair of
clauses, depending on the value (which is of mode 'boolean') of a condition,
e.g., \( (x > 0 \mid x \mid 0) \). Here, \( x > 0 \) is the condition. If its value is \texttt{true},
then \( x \), and, otherwise, 0 is chosen. Conditional-clauses are generalized in
the extensions 9.9, e.g., \( \texttt{if} x > 0 \texttt{then} x \texttt{else} x < -1 \texttt{then} -x - 1 \texttt{else} 0 \texttt{fi} \),
which has the same effect as \( (x > 0 \mid x \mid (x < -1 \mid -x - 1 \mid 0)) \). Unlike
similar constructions in other languages, conditional-clauses are always en-
closed between an \\texttt{if}-symbol, represented by \( \texttt{if} \), by \( \texttt{if} \) or by \( \texttt{case} \), and a \\texttt{fi}-
symbol, represented by \( \), by \( \texttt{fi} \) or by \( \texttt{case} \). This enclosure allows both parts
of the choice-clause and the condition to contain serial-clauses.)
6.4.1. Syntax

a) \textsf{SORTIVITY} conditional \textsf{CLAUSE}(24,55h,81d) : if symbol(31e), condition(b), \textsf{SORTIVITY} \textsf{choice} \textsf{CLAUSE}(c,d), fi symbol(31e).

b) condition(a) : strong serial boolean clause(61a).

c) \textsf{SORTIVITY} choice \textsf{CLAUSE}(a) :
\textsf{SORTIVITY} \textsf{then} \textsf{CLAUSE}(e), \textsf{SORTIVITY} \textsf{else} \textsf{CLAUSE}(e).

da) \textsf{FEAT} choice \textsf{CLAUSE}(a) :
\textsf{FEAT} \textsf{then} \textsf{CLAUSE}(e), \textsf{strong} else \textsf{CLAUSE}(e);
\textsf{strong} then \textsf{CLAUSE}(e), \textsf{FEAT} else \textsf{CLAUSE}(e).

e) \textsf{SORTIVITY THEELSE CLAUSE}(c,d) :
\textsf{THEELSE} symbol(31e), \textsf{SORTIVITY} serial \textsf{CLAUSE}(61a).

(Examples:

a) \((x > 0 \mid x \mid 0) \; \text{if} \; \text{overflow} \; \text{then} \; \text{exit} \; \text{fi} \) (see 9.4.a) ;

b) \(x > 0 \; ; \; \text{overflow} \; ; \)

c) \(x \; ; \; \text{then} \; \text{exit} \; ; \)

d) \(x \; ; 0 \; \text{in} \; (x > 0 \mid x \mid 0) + y \; ; \)

e) \(x \; ; 0 \; \text{then} \; \text{exit} \; )

(Rule d illustrates why modes should be balanced (see also 6.1.1.g and 6.2.1.e)). Thus, if a choice-clause is, say, firm, then at least one of its two constituent clauses must be firm, while the other may be strong. For example, in \((p \mid x \mid \sim) + (p \mid \sim \mid y)\), the conditional-clause \((p \mid x \mid \sim)\) is balanced by making \(x\) firm and \(\sim\) strong, whereas \((p \mid \sim \mid y)\) is balanced by making \(\sim\) strong and \(y\) firm. The example \((p \mid \sim \mid \sim) + y\) illustrates that not both may be strong, for otherwise the operator + could not be identified.)

6.4.2. Semantics

a) A conditional-clause is elaborated in the following steps:
Step 1: Its condition is elaborated;
Step 2: If the value of that condition is true (false) then the serial-clause of the then-clause (else-clause) of its choice-clause is considered;
Step 3: The considered clause is elaborated;
Step 4: The value, if any, of the conditional-clause, then is that of the clause elaborated in Step 3.

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

(Unitary-declarations provide the indication-defining occurrences of
mode-indications, e.g., string in mode string = [1 : 0 flex] char, and of
dyadic-indications, e.g., \( \nu \) in priority \( \nu = 2 \), the defining occurrences of
mode-identifiers, e.g., \( x \) in real \( x \), and the operator-defining occurrences
of operators, e.g., \( \text{abs} \) in \( \text{op abs} = (\text{int} \ a) \ \text{int} : (a < 0 \mid -a \mid a) \).
Declarations occur in declaration-preludes (6.1.1.b).

7.0.1. Syntax

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

   (Examples:
   a) mode string = [1 : 0 flex] char ; priority \( \nu = 2 \);
      int \( m = 4096 \); op := (real \( a, b \)) int : round \( a + round b \)

7.0.2. Semantics

A mode-identifier (operator) which was caused to possess a value by the
elaboration of a declaration containing the defining (operator-defining) oc-
currence of that mode-identifier (operator) is caused to possess an unde-
dined value upon termination or completion of the elaboration of the small-
est range (4.1.1.e) containing that declaration.

7.1. Declarers

(Declerers are built from the symbols int, real, bool, char and format,
with the assistance of certain other symbols as, e.g., long, ref, [ ],
struct, union and proc. A declarer specifies a mode, e.g., real specifies
the mode 'real'. A declarer is either a declarator or a mode-indication,
e.g., compl is a mode-indication and not a declarator. Declarers are clas-
sified as actual, formal or virtual depending on the kind of lower- and
upper-bounds which are permitted. Formal-declarers have the greatest free-
dom in this respect; e.g., \( [1 : n ] \) real, \( [1 : n \ flex ] \) real, \( [1 : \ flex ] \) real,
\( [1 : n \ either ] \) real, \( [1 : either ] \) real and \( [] \) real may all be formal, but
only the first two may be actual and only the last one may be virtual.)

7.1.1. Syntax

a)* declarer : VICTAL MODE declarer(b).
b) VICTAL MODE declarer(a, k, l, m, n, o, p, x, y, as, jj, 5we, 72a, 834a, 851b, c):
   VICTAL MODE declarer(c, d, e, l, m, n, o, p, x, y, c);
   MODE mode indication(42b).
c) VICTAL PRIMITIVE declarer(b, d) : PRIMITIVE symbol(31d).
d) VICTAL long INTREAL declarer(b, d) : long symbol(31d), VICTAL INTREAL declarer(c, d).
7.1.1. continued

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

e) VICTUAL structured with FIELDS declarator(b) :
   structure symbol[31d], VICTUAL FIELDS declarator(f,h,k) pack.

f) VICTUAL FIELDS and FIELD declarator(e,f) :
   VICTUAL FIELDS declarator(f,h,k), comma symbol[31e],
   VICTUAL FIELD declarator(h,k).

g) field declarator : VICTUAL FIELD declarator(h,k).

h) VICTUAL STOWED field TAG declarator(e,f) :
   VICTUAL STOWED declarer(b), STOWED field TAG selector(j).

i) field selector : FIELD selector(j).

j) MODE field TAG selector(h,k,852a) : TAG(302b,41c,d).

k) VICTUAL NONSTOWED field TAG declarator(e,f) :
   virtual NONSTOWED declarer(b), NONSTOWED field TAG selector(j).

{Examples:
  e) struct (string title, [1 : n] ref string pages, int price) ;
  f) string title, [1 : n] ref string pages, int price ;
  h) [2 : n] ref string pages ;
  j) title ;
  k) int price}

{Rules h and k, together with 1.2.1.r,s,t,u,v and h.1.1.c,d lead to an
infinity of production rules of the strict language, thereby enabling
the syntax to "transfer" the field-selectors (i) into the mode of structured
values, and making it ungrammatical to use an "unknown" field-selector in a
selection (§5.5.2). Concerning the occurrence of a given field-selector more
than once in a declarer, see 4.4.3, which implies that struct (real x, int x)
is not a (correct) declarer, whereas struct (real x, struct (int x, bool p) p)
is. Notice, however, that the use of a given field-selector in two different
declarers within a given reach does not cause ambiguity. Thus, mode cell =
struct (string name, ref cell next) and mode link = struct (ref link next, ref
cell value) may both occur in the same reach.}

1) VIRACT reference to MODE declarator(b) :
   reference to symbol[31d], virtual MODE declarator(b).

m) formal reference to STOWED declarator(b) :
   reference to symbol[31d], formal STOWED declarer(b).

n) formal reference to NONSTOWED declarator(b) :
   reference to symbol[31d], virtual NONSTOWED declarator(b).

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

{Rules 1, m and n imply that, for instance, ref [1 : either] real x may
be a formal-parameter (5.4.1.e), whereas ref ref [1 : either] real x may
not.}
7.1.1. continued 2

o) VICTAL ROWS structured with FIELDS declarator(b):
   sub symbol(31e), VICTAL ROWS rower(q,r), bus symbol(31e),
   VICTAL structured with FIELDS declarator(b).

p) VICTAL ROWS NONSTOWED declarator(b):
   sub symbol(31e), VICTAL ROWS rower(q,r), bus symbol(31e),
   virtual NONSTOWED declarator(b).

q) VICTAL row of ROWS rower(o,p,q):
   VICTAL row of rower(r), comma symbol(31e), VICTAL ROWS rower(q,r).

r) VICTAL row of rower(o,p,q):
   VICTAL lower bound(s,t,v), up to symbol(31e), VICTAL upper bound(s,t,v).

s) virtual LOWER bound(r) : EMPTY.

t) actual LOWER bound(r):
   strict LOWER bound(u), flexible symbol(31d) option.

u) strict LOWER bound(t,v,861f) : strong integral tertiary(81b).

v) formal LOWER bound(r):
   strict LOWER bound(u) option, flexible symbol(31d) option;
   strict LOWER bound(u) option, either symbol(31d).

{Examples:
 o) [I : m] struct ([I : n] real a, int b);
 p) [I : m, I : n] ref [] real;
 q) I : m, I : n;
 r) I : m;
 t) m ; m flex ;
 u) m ;
 v) m ; m flex ; flex ; m either ; either }

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

w) VICTAL PROCEDURE declarator(b):
   procedure symbol(31d), virtual PROCEDURE plan(x,aa).

x) virtual procedure with PARAMETERS MOD plan(w,75b):
   virtual PARAMETERS(y,54c) pack, virtual MOD declarer(b,z).

y) virtual MODE parameter(x,54c) : virtual MODE declarer(b).

z) virtual void declarer(x,aa,83ha) : EMPTY.

aa) virtual procedure MOD plan(w) : virtual MOD declarer(b,z).

bb) parameters pack : VICTAL PARAMETERS(y,54c,e,74b) pack.
7.1.1. continued 3

w) \text{proc } \text{proc} \ (<\text{real}, \text{int}) \ ; \ \text{proc} \text{ bool} \ ; \ \text{proc} \ (<\text{real}) \text{ bool} ;

x) \text{real} ; \text{int} ; \text{real} \text{ bool} ;

y) \text{bool} ;

z) \text{bool} ;

cc) \text{VIRTUAL union of } \text{LMOODS MOOD mode declarator(b)}:
   \text{union of symbol(31a), LMOODS MOOD and open box(dd) pack.}

dd) \text{LOSETY LMOOD open BOX(cc,ee) : LOSETY closed LMOOD end BOX(ee,ff).}

ee) \text{LOSETY closed LMOODSETY LMOOD end BOX(dd,ee,ff) :}
    \text{LOSETY closed LMOODSETY LMOOD end BOX(ee,ff) ;}
    \text{LOSETY open LMOODSETY LMOOD BOX(dd,ee).}

ff) \text{LOSETY closed LMOODSETY LMOOD end LMOOT BOX(dd,ee,ff):}
    \text{LOSETY closed LMOODSETY LMOOD end BOX(ee,ff).}


gg) \text{open LMOODS LMOOD BOX(ee,gg,ii) : LMOODS LMOOD BOX(ii) ;}
    \text{open LMOODS box(gg,hh), comma symbol(31e), LMOOD BOX(ii,jj).}

hh) \text{open LMOOD box(gg) : LMOOD BOX(jj).}

ii) \text{LMOODS MOOD and box(gg): union of LMOODS MOOD mode declaration(42b) ;}
    \text{union of symbol(31a), open LMOODS MOOD and box(gg) pack.}

jj) \text{MOOD and box(gg,hh) : virtual MOOD declarator(b).}

cc) \text{union (real, union (int, bool), union (real, int)) ;}
    \text{union (ri, union (bool, real)) (in the reach of union ri = (real, int))}

(\text{Let } "b" \text{ stand for 'boolean', } "i" \text{ for 'integral', } "r" \text{ for 'real', } "+" \text{ for 'and' and } "[bit]" \text{ for any of the six protonotions } 'b+i+r', 'b+r+i',
   'i+b+r', 'i+r+b', 'r+b+i' \text{ and 'r+i+b'. Each example is then examples of a virtual-, actual- or formal-union-of-(bit)-mode-declarator. The choice for}
   \text{bit} \text{ is left undefined and is semantically irrelevant, but if one chooses some canonical ordering of all mode involved in a program, then the rules}
   \text{cc up to jj and 8.2.4.1.a,b,c,d do not cause any ambiguity (1.1.6.1,}\text{4.4.3.d).}

The production mechanism of the rules cc up to jj is such that rule ee1 repeats, rule ff commutes and rule gg associates modes, whereas rule dd closes
and rule ee2 opens the box. Let "#" stand for 'box', "(" for 'closed', ")" for 'end!', "[" for 'open' and "]" for 'comma symbol.', then the production of the first example from 'actual union of integral and real and boolean
mode declarator' is suggested by:

cc \text{i+r+b} \text{i} \text{ee1} \text{i+r+k} \text{b} \text{ff} \text{r+i+b+k+l} \text{ee2} \text{[r+i+b+k+l+\#}

dd \text{i+r+k} \text{b} \text{ff} \text{i+r+k+\#} \text{ee1} \text{[i+r+k]b+r+\#} \text{gg} \text{[i+r+k+b+l+\#}

ee2 \text{i+r+k} \text{b} \text{ee2} \text{i+r+k+l+\#} \text{ff} \text{r+i+b+k+l+\#} \text{gg} \text{[r+i+b+k+l+\#}

da \text{i+r+k+l} \text{b} \text{da} \text{i+r+k+l+\#} \text{ff} \text{r+i+b+k+l+\#} \text{hh} \text{r+i+b+k+l+\#}

7.1.2. Semantics

a) A given declarer specifies the mode enveloped (1.1.6.j) by its original
   (1.1.6.c).

b) A given declarer is protected by protecting all its constituent serial-
   clauses.

c) A given declarer is "developed" in the following steps:
   Step 1: It is protected;
   Step 2: If it is, or contains and does not shield, a mode-indication which
7.1.2. continued

is an actual-declarer or formal-declarer, then that indication is replaced by a copy of the protected actual-declarer of that mode-declaration (7.2) whose mode-indication is its indication-defining occurrence (4.2.2.a), and Step 2 is taken; otherwise, the development of the declarer has been accomplished.

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

d) A (virtual- or actual-) declarer $D$ specifying the mode $M$ is elaborated in the following steps:
Step 1: If $D$ is a virtual-declarer, then a new instance $V$ of some value of some mode $N$ (not beginning with 'union of' and) such that $M$ is, or is united from, $N$ is chosen and Step 8 is taken; otherwise, $D$ is developed and Step 2 is taken;
Step 2: If $D$ now begins with a structure-symbol, then Step 4 is taken; otherwise, if $D$ now begins with a sub-symbol, then Step 5 is taken; otherwise, if $D$ now begins with a union-of-symbol, then Step 3 is taken; otherwise, a new instance $V$ of some value of the mode $M$ is chosen and Step 8 is taken;
Step 3: Some mode $N$ is chosen which does not begin with 'union of' and from which $M$ is united (4.4.3.a); a new instance $V$ of some value of the mode $N$ is chosen and Step 8 is taken;
Step 4: All constituent declarers of $D$ are elaborated collaterally (6.2.2.a); the values referred to by the values (names) of these declarers are made, in the given order, to be the fields of a new instance $V$ of a structured value of the mode $M$, and Step 8 is taken;
Step 5: All constituent boundscripts (6.6.1.1.1) of $D$ are elaborated collaterally;
Step 6: A descriptor (2.2.3.3) $Q$ is established consisting of an offset $I$ and as many quintuples, say $n$, as there are constituent actual-row-of-rowers in $D$; for $i = 1, \ldots , n$, $\ell_{i} \cdot (u_{i})$ is set equal to the value of the constituent strict-lower-bound (strict-upper-bound) of the $i$-th of these actual-row-of-rowers; if the flexible-symbol-option of the actual-lower-bound (actual-upper-bound) of the $i$-th of these actual-row-of-rowers is a flexible-symbol, then $\ell_{i} \cdot (t_{i})$ is set to 0; otherwise, $\ell_{i} \cdot (t_{i})$ is set to 1; next, $d_{i}$ is set to 1, and for $i = n, n - 1, \ldots , 1$, the stride $d_{i} - 1$ is set to $(u_{i} - \ell_{i} + 1) \times d_{i} ;$
Step 7: $Q$ is made to be the descriptor of a new instance $V$ of a multiple value of the mode $M$, whose elements are obtained as follows: the last constituent declarer of $D$ is elaborated collaterally a number of times and the elements are copies of values referred to by (some of the) resulting names;
Step 8: A name (2.2.3.5) different from all other names and whose mode is 'reference to' followed by $M$, is created and made to refer to $V$; this name is the value of $D$.

7.2. Mode declarations

_Mode-declarations provide the indication-defining occurrences of mode-indications, which act as abbreviations for declarers built from some basic-tokens, e.g., $mode \ string = [1 : 0 \ flex \ char$, or from other declarers or even from themselves, e.g., $mode \ book = struct \ (string \ title, ref \ book \ next$._ In this last example, the mode-indication is not only a convenient abbreviation, but it is essential to the declaration._}
7.2.1. Syntax

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

   (Examples:
   a) mode string = [1 : 0 flex] char ; struct compl = (real re, im);
   union primitive = (int, real, bool, char, format) (see 9.2.b,c)

7.2.2. Semantics

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

7.3. Priority declarations

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

7.3.1. Syntax

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

b) one token(a) : digit one symbol(31b).

c) TWO token(a) : digit two symbol(31b).
d) THREE token(a) : digit three symbol(31b).

e) FOUR token(a) : digit four symbol(31b).
f) FIVE token(a) : digit five symbol(31b).
g) SIX token(a) : digit six symbol(31b).
h) SEVEN token(a) : digit seven symbol(31b).

d) EIGHT token(a) : digit eight symbol(31b).
j) NINE token(a) : digit nine symbol(31b).

   (Example:
   a) priority + = 6 )

7.3.2. Semantics

   The elaboration of a priority-declaration involves no action.
   (For a summary of the standard priority-declarations, see the remarks in
    8.4.2.)
7.4. Identity-declarations

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

7.4.1. Syntax

a) identity declaration{70a} : formal MODE parameter{54e},
eesquals symbol{31c}, actual MODE parameter{b}.
b) actual MODE parameter{a,54c,75a,862a} :
strong MODE unit{61e} ; MODE transform{558a,-}.

\{Examples:\}
a) \texttt{real e = 2.718281828460045 ; int e = abs i ; real d = re (x \times conj x) ;}
\texttt{ref [.] real al = a [., k] ; ref real x1k = x1 [k] ; compl unit = 1 ;}
\texttt{proc int time = clock + cycales ;}
(The following declarations are given first without, and then with, the extensions of 9.2)
\texttt{ref real x = loc real ; real x ;}
\texttt{ref int sum = loc int := 0 ; int sum := 0 ;}
\texttt{ref (either : either, either : either) real a = loc [1 : m, 1 : n] real := x1 ; t := m, t := n ; real a := x2 ;}
\texttt{proc (real) real vera = ((real x) x := x1 := 1 - cos (a)) ;}
\texttt{proc vera = (real x) x := 1 - cos (a) ;}
\texttt{ref proc (int) int q = loc proc (int) int := ((int i) int := abs i) ;}
\texttt{proc q := (int j) int := abs i ;}
b) \texttt{abs i ; loc real ; loc int := 0 ;}$d.11de+2d$

7.4.2. Semantics

An identity-declaration is elaborated in the following steps:
Step 1: The \textit{formal-declarer} \( D \) of its \textit{formal-parameter} \( F \) is developed{7.1.2.a};
Step 2: Its \textit{actual-parameter} \( A \) and all boundscripts contained in \( D \), as possibly modified in Step 1, but not contained in any boundscript contained in \( D \), are elaborated collaterally{6.2.2.a};
Step 3: If the value \( V \) of \( A \) is a name which refers to a component{2.2.2.k} of a multiple value having one or more states equal to \( 0 \), then the further elaboration is undefined; otherwise, if \( V \) is a name other than \( nil \), then the value to which \( V \) refers, or otherwise \( V \) itself, is termed \( W \); Step 4: If \( W \) is not a structured value or multiple value, then Step 7 is taken; otherwise, if \( V \) is not a name, then Step 6 is taken;
Step 5: For each flexible-symbol-option \( S \) contained in \( D \), as possibly modified in Step 1, but not contained in any boundscript contained in \( D \), if the corresponding state is checked, \( \text{i.e.,} \) if \( S \) is a flexible-symbol (empty) and the corresponding state in \( W \) is \( 1 (0) \), then the further elaboration is undefined; otherwise, Step 6 is taken;
Step 6: For each boundscript contained in \( D \), as possibly modified in Step 1, but not contained in any boundscript contained in \( D \) and not followed by a flexible-symbol, \( \text{the corresponding bound is checked,} \text{i.e.,} \) if its value is not the same as the corresponding bound, if any, in \( W \), then the further elaboration is undefined; otherwise, Step 7 is taken;
7.4.2. continued

Step 7: The identifier of $F$ is made to possess $V$.

(According to Step 6, the elaboration of the declaration
\[ [1 : 2] \text{real} \ ax = (1.3, 3.4, 5.6) \]
is undefined and according to Step 5 the elaboration of the declaration
\[ \text{ref}[1 : \text{false}] \text{real} \ ax = [1 : 2] \text{real} := (1.2, 3.4) \]
is undefined. The elaboration of the declaration
\[ [1 : \text{flags}] \text{real} \ ax = (1.2, 3.4) \]
is well defined, but its effect is also obtained by the elaboration of the
less confusing declaration
\[ [] \text{real} \ ax = (1.3, 3.4). \]

7.5. Operation declarations

Operation-declarations provide the operator-defining occurrences of
operators, e.g., \( \text{op} \lor = (\text{real} \ a, b) \text{real} : (\text{random} < .5 \mid a \mid b) \), which con-
tains an operator-defining occurrence of \( \lor \) as a dyadic-operator. Unlike
identity-declarations of which no two for the same identifier may occur in
a reach (4.4.2.b), more than one operation-declaration involving the same
adic-indication may occur in the same reach, see 10.2.3.i, 10.2.4.i, etc.)

7.5.1. Syntax

a) operation declaration(70a):
   \( \text{PRAM} \) caption(b), equals symbol(31c), actual PRAM parameter(74b).
b) PRAM caption(a) : operation symbol(31d),
   virtual PRAM plan(71x), PRAM ADIC operator(43b,c,-).

   (Examples:
   a) \( \text{op} \land = (\text{bool} \ a, b) \text{bool} : (a \mid b \mid \text{false}) ; \)
      \( \text{op} \leftrightarrow (\text{real} \ a) \text{real} : (a < 0 \mid -a \mid a) \) (see 9.2.d,e) ;
   b) \( \text{op} (\text{bool}, \text{bool}) \text{bool} \land \); \( \text{op} (\text{real}) \text{real} \leftrightarrow \)

7.5.2. Semantics

An operation-declaration is elaborated in the following steps:
Step 1: Its actual-parameter is elaborated;
Step 2: The operator of its caption is made to possess the (routine which
   is) value obtained in Step 1.

(The formula (8.4.1) \( p \land q \), where \( \land \) identifies the operator-defining
occurrence of \( \land \) in the operation-declaration
\( \text{op} \land = (\text{bool} \ a, b) \text{bool} : (\text{john} \mid \text{mccarthy} \mid \text{false}), \)
possesses the same value as it would if \( \land \) identified the operator-defining
occurrence of \( \land \) in the operation-declaration
\( \text{op} \land = (\text{bool} \ a, b) \text{bool} : (a \mid b \mid \text{false}), \)
except, possibly, when the elaboration of \( q \) involves side effects on that
of \( p \).)
8. Unitary clauses

(Unitary-clauses may occur as actual-parameters, e.g., \( z \) in \( \sin (z) \), as sources in assignations, e.g., \( y \) in \( z := y \), in casts, especially in routine-denotations, e.g., \( i := 1 \) in \( ((\text{ref int } i) \text{ int } : i := 1) \), or may be used to construct serial-clauses or collateral-clauses, e.g., \( z := 1 \) in \( (x := 1; y := z) \) or in \( (x := 1, y := z) \). Unitary-clauses either are closed, collateral- or conditional, or are coercends. There are four kinds of coercends: confrontations, e.g., \( z := 1 \), formulas, e.g., \( x + 1 \), cohesions, e.g., \( \text{next of cell} \), and bases, e.g., \( x \). These coercends and the closed-, collateral- and conditional-classes are grouped into the following four classes, each class being a subclass of the next: primaries, which may be subscipted and parametrized, e.g., \( x! \) and \( \sin (z) \); secondaries, from which fields may be selected, e.g., \( z \) in \( \text{re } z \); tertiaries, which may be operands, or may be destinations in assignations, or may occur in identity- or conformity-relations, e.g., \( x \) in \( x + 1 \) or in \( x := 1 \) or in \( x := y \) or in \( x := \text{tr} \), or may be boundscripts, e.g., \( m \), \( 0 \) and \( n \) in \( x \) [: \( m \equiv 0, n \)], and, finally, unitary-clauses, which is the largest class. Thus, \( r \) of \( s \) \((t)\) means that \( s \) is first called or subscripted, whereas \( (r \) of \( s \) \((t)\)) means that the field is selected first. Also, \( r \) of \( s + t \) means that the field is selected from \( s \) before elaborating the routine possessed by \( + \), while to force the elaboration of \( + \) first, one must write \( r \) of \( (s + t) \).

8.1.1. Syntax

a) \( \text{SORTUNITARY } \text{MOID clause}(61e) : \text{SORTUNITARY } \text{MOID tertiary}(b) \); 
   \( \text{SORTUNITARY } \text{MOID confrontation}(820d,e,f,g,830a,\ldots) \).
b) \( \text{SORTUNITARY } \text{MOID tertiary}(a,71u,831b,832a,833a,861b,\ldots) : \text{SORTUNITARY } \text{MOID secondary}(c) \); 
   \( \text{SORTUNITARY } \text{MOID ADIC formula}(820d,e,f,g,86b,\ldots) \).
c) \( \text{SORTUNITARY } \text{MOID secondary}(b,84k,852a) : \text{SORTUNITARY } \text{MOID primary}(d) \);
   \( \text{SORTUNITARY } \text{MOID cohesion}(820d,e,f,g,850a) \).
d) \( \text{SORTUNITARY } \text{MOID primary}(c,861a,862a) : \text{SORTUNITARY } \text{MOID base}(820d,e,f,g,860a,b) \); 
   \( \text{SORTUNITARY } \text{CLOSED } \text{MOID clause}(62b,c,d,f,63a,64a,\ldots) \).

{Examples:}
a) \( z ; z := 1 \); 
b) \( z ; z + 1 \); 
c) \( z ; \text{real} \); 
d) \( z ; (z + 1) \)

8.2. Coercends

{Coercends are of four kinds: bases, e.g., \( x \), cohesions, e.g., \( \text{re of } x \), formulas, e.g., \( x + y \) and confrontations, e.g., \( x := 1 \). These are collectively considered as coercends because it is in their production rules that the basic coercions appear.

In current programming languages certain implicit changes of type are described, usually in the semantics. Thus \( z := 1 \) may mean that the integral value of \( z \) yields an equivalent real value which is then assigned to the name possessed by \( x \). In ALGOL 68, such implicit changes of mode are known as coercions, and are reflected in the syntax. Certain coercions available in other languages, such as that in \( x \) := \( z \), are not permitted. One must write \( i := \text{round } x \) or \( i := \text{entier } x \), for in this situation it is felt advisable
8.2. continued

for the programmer to state the coercion explicitly. Apart from this, all
the coercions which the programmer might reasonably expect are supplied.

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

The kinds of coercion which are used depend upon three things: "syntactic
position", a priori mode and a posteriori mode (i.e., the modes before
and after coercion). There are four sorts of syntactic position. They are:
"strong" positions, i.e., actual-parameters, e.g., \( x \) in \( \text{in } (x) \), sources,
e.g., \( x \) in \( y := x \), conditions, e.g., \( x > 0 \) in \( (x > 0 \mid x < 0) \), subscripts,
e.g., \( i \) in \( x[l] \), etc.; "firm" positions, i.e., operands, e.g., \( x \) in \( x + y \),
transforms, e.g., \( $5d$ \), and certain primaries, e.g., \( x \in \in (x) \);
"weak" positions, i.e., certain primaries, e.g., \( x[l] \) in \( x[l] \), and certain
secondaries, e.g., \( z \) in \( \text{re of } z \); and "soft" positions, i.e., destinations,
e.g., \( x \) in \( x := y \), and some other secondaries, e.g., \( x \) in \( \text{ax} := x \).

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

In the productions of a notion, the sort (strong, firm, weak, soft) of
position is passed on, or modified during balancing (to strong), and leads
to basic coercions which appear in the production rules for coercers; moreover,
the coercion must be completely expanded in these rules. For example,
\( y \) in \( x := y \) is a real-source and, therefore, a strong-real-unit (8.3.1.1.0);
the sort 'strong' is passed through the productions of 'strong real unit'
until a 'strong real base' is reached (8.1.3.d); this is then produced to
'strongly dereferenced to real base' (8.2.0.1.0), next to 'reference to real
base' (8.2.1.1.0) and finally to 'reference to real mode identifier'
(8.0.0.0.1.0).
8.2.0.1. Syntax

a)* coerced : SORT COERCEND(d,e,f,g) ; SORTLY ADAPTED to COERCEND(821a,b,822a,b,c,823a,824a,825a,b,c,d,826a,827a,828a,b,-).
b)* SORT coerced : SORT COERCEND(d,e,f,g).
c)* ADAPTED coerced : SORTLY ADAPTED to COERCEND(821a,b,822a,b,c,823a,824a,825a,b,c,d,826a,827a,828a,b,-).
d) strong COERCEND(81a,b,c,d) : COERCEND(830a,84b,g,850a,860a,b,-) ; strongly ADAPTED to COERCEND(821a,b,822a,b,c,823a,824a,825a,b,c,d,826a,827a,828a,b,-).
e) firm COERCEND(81a,b,c,d,84d,f) : COERCEND(830a,84b,g,850a,860a,b,-) ; firmly ADJUSTED to COERCEND(821a,b,822a,b,c,824a,-).
f) weak COERCEND(81a,b,c,d) : COERCEND(830a,84b,g,850a,860a,b,-) ; weakly FITTED to COERCEND(821b,822b,-).
g) soft COERCEND(81a,b,c,d) : COERCEND(830a,84b,g,850a,860a,b,-) ; softly deprocedured to COERCEND(822c).

(Examples:

a) 3.14 (in x := 3.14) ; y (in x := y) ;
b) 3.14 ; x (in x := 3.14 + x) ; sin (in sin (x)) ;
c) x1 (in x1 [x1]) ; xx (in re of xx in the reach of ref compt xx) ;
d) x (in x := 1) ; x or y (in x or y := 3.14) ;

8.2.1. Dereferenced coerceds

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

8.2.1.1. Syntax

a) STIRLily dereferenced to MODE FORM(a,820a,e,822a,b,823a,824b,d,825a,b,c,d,826a) : reference to MODE FORM(830a,84b,g,850a,860a) ;
   STIRLily FITTED to reference to MODE FORM(a,822a).
b) weakly dereferenced to reference to MODE FORM(b,820f) :
   reference to reference to MODE FORM(830a,84b,g,850a,860a) ;
   weakly FITTED to reference to reference to MODE FORM(b,822b).

(Examples:

a) y (in x := y or in x + y) ; yy (in x := yy or in x + yy) ;
b) rm1 (in rm1 [x] in the reach of ref [] real, rm1) ;

8.2.1.2. Semantics

A dereferenced-coerced is elaborated in the following steps:
Step 1: If the value obtained in Step 1 is not nil, then the value of the dereferenced-coerced is a copy of the value referred to by the value (name) obtained in Step 1; otherwise, the further elaboration is undefined.
8.2.1.2. continued

(Weak dereferencing must look ahead so that it does not remove a 'reference to' which precedes a mode which does not begin with 'reference to'. For example, in \( zl \[ \{ t \} := y \), the primary \( zl \) should not be dereferenced, for \( zl \[ \{ t \} + y \), the \( zl \) is not dereferenced but the base \( zl \[ \{ t \} \) is.)

8.2.2. Deprocedured coercends

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

8.2.2.1. Syntax

\begin{itemize}
\item[a)] STURMLy deprocedured to MDID FORM\{a, b, 820d, e, 821a, 824b, d, 825a, b, c, d, 826a, 826b\} ; procedure MDID FORM\{830a, 84b, 850a, 860a\} ; STURMLy FITTED to procedure MDID FORM\{a, 821a\}.
\item[b)] weakly deprocedured to MODE FORM\{820r, 821b\} ; procedure MODE FORM\{830a, 84b, 850a, 860a\} ; firmly FITTED to procedure MODE FORM\{a, 821a\}.
\item[c)] softly deprocedured to MODE FORM\{c, 820g\} ; procedure MODE FORM\{830a, 84b, 850a, 860a\} ; softly deprocedured to procedure MODE FORM\{c\}.
\end{itemize}

(Examples:
\begin{itemize}
\item[a)] random (in \( x := \text{random} \) or in \( x + \text{random} \) ;
\item[b)] \( r z \) (in \( r z \) of \( r z \) in the reach of \( \text{proc compl} r z = \text{compl} : (\text{random}, \text{random}) \) ;
\item[c)] \( x \) or \( y \) (in \( x \) or \( y := 1 \) )
\end{itemize}

8.2.2.2. Semantics

A deprocedured-coercend is elaborated in the following steps:
Step 1: It is preelaborated \{1.1.6.i\};
Step 2: The deprocedured-coercend is replaced by a closed-clause which is a copy of (the routine which is) its prevalue obtained in Step 1, and the elaboration of that closed-clause is initiated; the value yielded, if any, is that of the deprocedured-coercend and if this elaboration is completed or terminated, then the closed-clause is replaced by the deprocedured-coercend before the elaboration of a successor is initiated.
(See also calls, 8.6.2.)
8.2.3. Procedured coercends

(Coercends are procedured when it is required that an initial 'procedure' should be placed before the a priori mode (i.e., they should be turned into procedures without parameters), e.g., \( x := 1 \) in \( \text{proc real } p := x := 1 \). Here, 1 is not assigned to \( x \), but that routine which assigns 1 to \( x \) is assigned to \( p \). Notice, that \( \text{proc } p := x := 1 \) is syntactically incorrect, since \( x := 1 \) must first be voided before it can be procedured to the mode 'procedure void'; the way to achieve this is by using a void-cast-pack: \( \text{proc } p := (\dot{x} := 1) \). For the coercion in \( \text{proc stop = exit see 8.2.7} \).)

8.2.3.1. Syntax

a) \text{ST}IM\text{PLY} procedured to procedure \text{MOID FORM}(a,820a,e,824b,826a) :
   \text{MOID FORM}(830a,86b,g,850a,860a,b,-) ;
   \text{ST}IM\text{PLY} dereferenced to \text{MOID FORM}(821a,-);
   \text{ST}IM\text{PLY} procedured to \text{MOID FORM}(a,-);
   \text{ST}IM\text{PLY} united to \text{MOID FORM}(82a,-);
   \text{ST}IM\text{PLY} widened to \text{MOID FORM}(825a,b,c,d,-);
   \text{ST}IM\text{PLY} rowed to \text{MOID FORM}(826a,-).

(Examples:
   a) 3.14 (in \text{proc real } p := 3.14) ; x (in \text{proc real } p := x) ;
   3.14 (in \text{proc proc real } p := 3.14);
   1 (in \text{proc union } (\text{int, real } p := 1) ;
   \text{int} (in \text{proc } \int p := 1) )

8.2.3.2. Semantics

A procedured-coercend is elaborated in the following steps:
Step 1: A copy is made of it [itself, not its value]; if the mode enveloped by the original of the procedured-coercend is 'procedure' followed by a second mode (not by 'void'), then the second mode is considered; otherwise, Step 3 is taken;
Step 2: A virtual-declarer, which, if it occurred in the smallest reach containing the procedured-coercend, would specify the considered mode, followed by a cast-of-symbol is placed before the copy;
Step 3: An open-symbol is placed before and a close-symbol is placed after the copy as possibly modified in Step 2; the thus modified copy is the (routine which is the) value of the procedured-coercend.

(The elaboration of the strong-procedure-real-base (p | x1 | y1) [i] yields the routine (real : (p | x1 | y1) [i]), whereas that of the strong-conditional-procedure-real-clause (p | x1[i]) | y1[i]) yields either the routine (real : x1[i]) or the routine (real : y1[i]) depending on the value of p; similarly, the elaboration of the firm-procedure-real-confrontation \( x := (a := a + 1; y) \) yields the routine (real : x := (a := a + 1; y)), whereas that of the firm-closed-procedure-real-clause (a := a + 1; x := y) yields, apart from a change in the value of a, the routine (real : x := y); as last example, the elaboration of the strong-procedure-void-base (i : x := i + 1) yields the routine (i : x := i + 1).)
8.2.4. United coerced

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

8.2.4.1. Syntax

a) STRONGLY united to union of LMoods Mood mode FORM(8204,e,823a,826a) :
   one out of LMoods Mood mode FORM(b);
   some of LMoods Mood and but not FORM(c).

b) one out of LMoods Mood RMoodsity mode FORM(a) :
   Mood FORM(830a,84b,g,850a,860a);
   firmly fitted to Mood FORM(821a,822a);
   firmly proceeded to Mood FORM(823a,-).

c) some of LMoodsity Mood and RMoodsity but not LOSSITY FORM(a,c) :
   some of LMoodsity and Mood RMoodsity but not LOSSITY FORM(c,d,-);
   some of LOSSITY RMoodsity but not Mood and LOSSITY FORM(c,d,-).

d) some of and LMoods Mood RMoodsity but not LOSSITY FORM(c) :
   union of LMoods Mood RMoodsity mode FORM(830a,84b,g,850a,860a);
   firmly fitted to union of LMoods Mood RMoodsity mode FORM(821a,822a).

{Examples:

a) 2; \textit{tr} ;
b) 2; i; \textit{true} ;
d) \textit{tr} ; \textit{ir} (all in (union \textit{ir} = (int, real); \textit{ir} := 2; \textit{ir} \textit{r} = (p | i | w);
   union (\textit{ir}, \textit{proc bool}) \textit{tr} := \textit{r}; \textit{tr} := \textit{ir}; \textit{trb} := \textit{true} ))}

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

8.2.5. Widened coerced

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

8.2.5.1. Syntax

a) strongly widened to LONGERITY real FORM(b,8204,823a,826a) :
   LONGERITY integral FORM(830a,84b,g,850a,860a);
   strongly fitted to LONGERITY integral FORM(821a,822a).

b) strongly widened to structured with REAL field letter \textit{r} letter e
   and REAL field letter \textit{l} letter \textit{m} FORM(8204,823a,826a) :
   REAL FORM(830a,84b,g,850a,860a);
   strongly fitted to REAL FORM(821a,822a);
   strongly widened to REAL FORM(a).

c) strongly widened to row of boolean FORM(8204,823a,826a) :
   BITS FORM(830a,84b,g,850a,860a);
   strongly fitted to BITS FORM(821a,822a).
8.2.5.1. continued

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

   (Examples:
   a) 1 (in x := 1) ; i (in z := i) ;
   b) 3.14 (in x := 3.14) ; x (in z := x) ; I (in z := I) ;
   c) 1 0 1 ; t (in [1 : 3] bool b1 := (p | i 0 1 | t)) ;
   d) atb "abo" ; r (in s := [p | atb "abo" | r]) )

8.2.5.2. Semantics

A widened-coercend is elaborated in the following steps:
Step 1: It is preelaborated (1.1.6.1) and the value yielded is considered;
Step 2: If the considered value is an integer, then the real number equiv- 
   alent to it (2.2.3.1.d) is considered instead; otherwise, if the considered
   value is a real number, then the structured [complex (10.2.7)] value com- 
   posed of two fields, which are the considered value and the real number 0
   of the same length number as that of the considered value and which are
   selected by letter-r-letter-e and letter-i-letter-m respectively is con- 
   sidered instead; otherwise, (the considered value is a structured value
   with one field and) the field of the considered value is considered in-
   stead;
Step 3: The value of the widened-coercend is a new instance of the consid- 
   ered value; its mode is that enveloped by the original of the widened-
   coercend.

   (Widening may not be done in firm positions, for, otherwise, x := i + 1
   might be ambiguous.)

8.2.6. Rowed coercends

   (Coercends are rowed when it is required that 'row of' should be placed
   either before the a priori mode or after an initial 'reference to' of the a
   priori mode; e.g., in [I : J] real a1 := 3.14, the a priori mode of the base
   3,14 is 'real' but the a posteriori mode required in this strong position is
   'row of real', whereas in ref [I : J] real a2 := x, the a priori mode of the
   base x is 'reference to real' but the a posteriori mode required is 'refer- 
   ence to row of real'. Here, the value to which x refers, is turned into a
   multiple value with a descriptor. Note that the value of a2 [I] := ; x is
   true.)

8.2.6.1. Syntax

a) strongly rowed to REFETY row of MODE FORM(a,820d,823a) :
   REFETY MODE FORM(830a,84b,g,850a,860a) ;
   strongly ADJUSTED to REFETY MODE FORM(821a,822a,823a,824a,7) ;
   strongly widened to REFETY MODE FORM(825a,b,c,d,7) ;
   strongly rowed to REFETY MODE FORM(a,7) ;
   REFETY row of MODE FORM vacuum(b,7).

b) row of NONROW base vacuum(a) : EMPTY.
8.2.6.1. continued

(Examples:

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

8.2.6.2. Semantics

A rowed-coercerd is elaborated in the following steps:

Step 1: The mode enveloped by the original of the rowed-coercerd is considered; if that mode begins with 'row of row of' or if the rowed-coercerd is not empty, then it is preelaborated (1.1.6.i), the value obtained and its scope are considered and Step 3 is taken;

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

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

Step 4: That instance of the value to which the (name which is the) considered value refers is considered instead; if the considered value is a multiple value having one or more states equal to \texttt{0}, or if it is a component (2.2.2.k) of such a multiple value, then the further elaboration is undefined; otherwise, Step 5 is taken;

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

Step 6: Let \(d\) stand for \([(u_1 - \ell_1 + 1) \times d_1]\); the instance of a new multiple value, composed of the elements of the considered value and of a descriptor which is a copy of the descriptor of the considered value into which the additional quintuple \((1,1,0,1,1)\) is inserted before the first quintuple, and in which all states have been set to \texttt{1}, is considered instead;

Step 7: If the considered mode does not begin with 'reference to', then the value of the rowed-coercerd is the considered value; otherwise, a name \texttt{N} is made to refer to the considered value (whose scope is the prescope obtained in Step 1, and) whose mode is the considered mode; this name \texttt{N} is the value of the rowed-coercerd.

8.2.7. Hipped coercerds

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

a) strongly hipped to MOID base(820d) :
   MOID skip(b); MOID jump(c); MOID nihil(d,-).
b) MOID skip(a); skip symbol(31g).
c) MOID jump(a); go to symbol(31f) option, label identifier(41b).
d) reference to MOID nihil(a); nihil symbol(31g).

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

8.2.7.2. Semantics

a) A skip is elaborated in the following steps:
Step 1: If the terminal production of the metanotion 'MOID' enveloped
{1.1.6.j} by the original {1.1.6.c} of the skip is a mode, then this mode
is considered and Step 2 is taken; otherwise, {it is 'void' and} the elab-
oration of the skip is complete;
Step 2: If the considered mode begins with 'union of', then some mode from
which it is united {4.4.3.a} is considered instead;
Step 3: The value of the skip is a new instance of some value of the consid-
ered mode and whose scope is the program.

b) A jump is elaborated in the following steps:
Step 1: If the original of the jump envelops a mode which is 'procedureMOID'
where "MOID" stands for any terminal production of the metanotion 'MOID',
then this mode is considered and Step 2 is taken; otherwise, the elab-
oration of the jump is terminated and it appoints as its successor the
unitary-clause following the label-sequence or the completer containing
the defining occurrence (in a label (4.1.2)) identified by the label-
identifier of the jump;
Step 2: A copy is made of the jump and an open-symbol followed by a cast-of-
symbol is placed before and a close-symbol is placed after the copy; if
the considered mode is not 'procedure void', then the initial 'procedure'
is deleted from it and a virtual-declarer, which, if it occurred in the
smallest reach containing the jump, would specify the mode so obtained, is
inserted between the open-symbol and the cast-of-symbol in the copy;
otherwise, an open-symbol is placed before and a close-symbol is placed
after the thus modified copy;
Step 3: The value of the jump is the routine consisting of the same sequence
of symbols as the copy as modified in Step 2 and whose mode is that envel-
oped by the original of the jump.

c) The elaboration of a nihil involves no action; its value is a new in-
stance of nihil (2.2.3.5.a) whose mode is that enveloped by the original of
the nihil.

{Skips play a role in the semantics of routine-denotations (5.4.2.Step2)
and calls (8.6.2.2.Step 4). Moreover, they are usefull in a number of pro-
gramming situations, like e.g.,
supplying an actual-parameter (7.4.1.b) whose value is irrelevant or is to}
8.2.7.2. continued

be calculated later; e.g., \( f (3, x) \) where \( f \) does not use its second
actual-parameter if the value of the first actual-parameter is positive;
see also 11.11.ar;

supplying a constituent unit of a collateral-clause (6.2.1.b,c,d,f), e.g.,
\([1 : 4] \text{ real } e1 := (3.1d, \text{ skip}, 1.88, \text{ skip})\);
as a dummy statement (6.0.1.c) in those rare situations where the use of a
completer is inappropriate, e.g., \( l : \text{ skip} \) in 10.4.c. See also 9.4.a.

A jump is useful as a clause to terminate the elaboration of another
clause when certain requirements are not met, e.g., \text{ go to exit in }
y := \begin{cases} \text{ if } x \geq 0 \text{ then } \sqrt{x} (x) \text{ else } \text{ go to exit } \end{cases}.

If \( e1, e2 \) and \( e3 \) are label-identifiers, then the reader might recognize
the effect of the declaration \([]\) \text{ proc switch } = (e1, e2, e3) \) and the statement
\text{ switch } \{4\}; however, the declaration \([1 : 3 \text{ flex} \) \text{ proc switch } = (e1, e2, e3) \)
is perhaps more powerful, since assignments like \text{ switch } \{2\} := e1\) and
\text{ switch } := (e1, e2, e3, e4) \) are possible.

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

8.2.8. Voided coercends

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

8.2.8.1. Syntax

a) strongly voided to void confrontation(820a) : MODE confrontation(830a).
b) strongly voided to void FORESEE(820a) :
   NONPROC FORESEE(88b,e,860a,860a).
   strongly deprocedured to NONPROC FORESEE(822a).

[Examples:
 a) \( x := 1 \) (in \( (x := 1; y := 2) \) )
 b) \( x \); \text{ random} (in \( (e1; \text{ random}; \text{ skip}) \) )

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

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

8.3.0.1. Syntax

a) \texttt{MODE confrontation}(81a,82a,82d,e,f,g,82a,b,82a,b,c,82b,82b,d,82c,b,c, 
d,82a,82a) ; \texttt{MODE assignment}(83a,-) ; \texttt{MODE conformity relation} 
{83a,-} ; \texttt{MODE identity relation}(83a,-) ; \texttt{MODE cast}(83a).

(Examples:
 a) \texttt{x := 3.14} ; \texttt{ea := a (see 11.11.1)} ; \texttt{xx := x or y} ; [ ] \texttt{real} : I )

8.3.1. Assignations

(In assignations, e.g., \texttt{x := 3.14}, (an instance of) a value is assigned to a name. In \texttt{x := 3.14}, the value possessed by the source 3.14 is assigned to the (name which is the) value possessed by \texttt{x}.)

8.3.1.1. Syntax

a) reference to \texttt{MODE assignment}(830a) : reference to \texttt{MODE destination}(b), 
        becomes symbol(3c), \texttt{MODE source}(c). 
b) reference to \texttt{MODE destination}(a) : soft reference to \texttt{MODE tertiary}(81b). 
c) \texttt{MODE source}(a) : strong \texttt{MODE unit}(61c).

(Examples: 
 a) \texttt{x := 1} ; \texttt{loc real := 3.14} ; 
 b) \texttt{x} ; \texttt{loc real} ; 
 c) \texttt{1} ; 3.14)

8.3.1.2. Semantics

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

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

c) An instance of a value is assigned to a name in the following steps:
Step 1: If the given value does not refer to a component of a multiple value having one or more states equal to 0 (2.2.3.3.b), if the scope of the given name is not larger than the scope of the given value (2.2.4.2) and if the given name is not \texttt{null}, then Step 2 is taken; otherwise, the further elaboration is undefined;
Step 2: The instance of the value referred to by the given name is considered; if the mode of the given name begins with 'reference to structured with' or with 'reference to row of', then Step 3 is taken; otherwise, the considered instance is superseded [a] by a copy of the given instance and the assignment has been accomplished;
8.3.1.2. 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, if for some \( i, \ell, \ldots, n, d_i = 1 (t_i = 1) \) and \( t_i (u_i) \) is not equal to the corresponding bound in the descriptor of the given value, then the further elaboration is undefined;

Step 4: If some \( d_i = 0 \) or \( t_i = 0 \), then, first, a new instance of a multiple value \( M \) is created whose descriptor is a copy of the descriptor of the given value modified by setting its states to the corresponding states in the descriptor of the considered value, and whose elements are copies of elements, if any, of the considered value, and, otherwise, are new instances of values whose mode is, or is a mode from which is united, the mode obtained by deleting all initial 'row of' s from the mode of the considered value; next, \( M \) is made to be referred to by the given name and is considered instead;

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

d) An assignment is elaborated in the following steps:
Step 1: Its destination and source are elaborated collaterally (6.2.2.a);
Step 2: The value of its source is assigned to the (name which is the) value of its destination;
Step 3: The value of the assignment is the value of its destination.

(Observe that \((x, y) := (1, 2, 3, 4)\) is not an assignment, since \((x, y)\) is not a destination; the mode of the value of a collateral-clause (6.2.1. c,d,f) does not begin with 'reference to' but with 'row of' or 'structured with'.)

8.3.2. Conformity relations

(The purpose of conformity-relations is to enable the programmer to find out the current mode of an instance of a value if the context permits this mode to be one of a number of given modes. See, for example, 11.11.i,q,y,ah. Conformity-relations are thus used in conjunction with unions.)

(I would to God they would either conform,
or be more wise, and not be caught!
Diary, 7 Aug. 1664, Samuel Pepys.)

8.3.2.1. Syntax

a) boolean conformity relation(830a):
   soft reference to TMDTE tertiary(81b), conformity relator(b),
   RVMDTE tertiary(81b).

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

   [Examples:
   a) \( \text{int} := \text{int} \); \( \text{ea} := \text{a} \) (see 11.11.i) ;
b) \( \text{::} \); \( \text{::=} \) ]
8.3.2.2. Semantics

A conformity-relation is elaborated in the following steps:

Step 1: Its textually last tertiary is elaborated and the value yielded is considered;

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

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

Step 4: If its conformity-relator is a conforms-to-and-becomes-symbol, then its textually first tertiary is elaborated and the considered value is assigned (8.3.1.2.c) to the value of that tertiary.

(Although not suggested by the wording of Step 2, the, possibly, most obvious applications of conformity-relations are those in which 'RMODE' in 8.3.2.1.a begins with 'union of' whereas 'LMODE' does not. Then, the mode of the considered value (Step 1) is not 'RMODE' (which is united from it) and the conformity-relation serves to ask whether this mode is 'LMODE' and, if so and if the conformity-relator is a conforms-to-and-becomes-symbol, to assign this value to a name whose mode does not begin with 'reference to union of' and, thereby, make this value easily available elsewhere. Several applications, partly disguised by the application of the extensions 9.4 are given in 11.11.

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

8.3.3. Identity relations

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

8.3.3.1. Syntax

a) boolean identity relation(830a):

   soft reference to MODE tertiary(81b), identity relator(b),
   strong reference to MODE tertiary(81b);
   strong reference to MODE tertiary(81b), identity relator(b),
   soft reference to MODE tertiary(81b).

b) identity relator(a) : is symbol(31c) ; is not symbol(31c).
8.3.3.1. continued

{Examples:
a) \(x \ or \ y := x; \ xx := x;\)
b) \(:=; \.value\)

8.3.3.2. Semantics

An identity-relation is elaborated in the following Steps:
Step 1: Its tertiaries are elaborated collateral (6.2.2.a);
Step 2: If its identity-relator is an is-symbol (is-not-symbol), then the
value of the identity-relation is \textit{true} (false) if the (names which are
the) values obtained in Step 1 are the same and \textit{false} (true) otherwise.

(Assuming the assignment \(xx := yy := x\) to have been elaborated, the
value of the identity-relation \(xx := yy\) is \textit{false} because \(xx\) and \(yy\), though
of the same mode, do not possess the same name (7.1.2.Step 8), but the name
which each possesses refers to the same name and so \(\texttt{(ref real : xx)} :=\)
\(\texttt{(ref real : yy)}\) possesses the value \textit{true}. The value of the identity-relation
\(xx := x \ or \ y\) has a probability \(\frac{1}{2}\) of being \textit{true} because the value possessed
by \(xx\) (effectively that of \(\texttt{ref real : xx}\) here, because of coercion) is the
name possessed by \(xx\), and the routine possessed by \(x \ or \ y\) (see 1.3), when
elaborated, yields either the name possessed by \(x\) or, with equal proba-

In the identity-relation, the programmer is usually asking a specific
question concerning names and thus the level of reference is of crucial im-
portance. Thus at least one of the tertiaries of an identity-relation must
be soft, i.e., must involve only deproceduring and certainly no derefer-
encing. The construction \texttt{case i in x, xx, x or y, nil esac := case j in y,
skip, x or y, rest of z cut yy esac} is an example of a delicately balanced
identity-relation in which the mode is 'reference to real'.

Observe that the value of the formula \(j = 3\) is \textit{false}, whereas \(1 :=;\ 2\) is
not an identity-relation, since the values of its tertiaries are not names.
Also \texttt{$$2d*$$ := $$6d$$} is not an identity-relation, whereas \texttt{$$2d*$$ = $$6d$$}
is a formula, but involves an operation which is not included in the standard-
prelude.)

8.3.4. Casts

{Casts may be used to provide a strong position for a unitary-clause in
a position which is not strong, e.g., \texttt{ref real : xx in (ref real : xx) := 1.}
They play a role in routine-denotations (5.4.1.a), e.g., \texttt{real : a + 1 in
(tint a) real : a + 1} and procedure-coercions (8.2.3.1.a), e.g., \(:(1:1)
in proc busy = :: \(:(1:1)\). A void-cast is not a clause but is a constituent
of a void-cast-pack and of some routine-denotations and thus of bases.)

8.3.4.1. Syntax

a) \texttt{MOID cast(5b,830a,860b) : virtual MOID declarer(71b,z),
cast of symbol(31b), strong MOID unit(61e).}

{Examples:
a) \[[] \ real : 1 ; : x := 3.14 \]}

8.3.4.2. Semantics

The elaboration (value, if any,) of a cast is that of its unit.
8.4. Formulas

(Formulas are either dyadic-formulas, e.g., \( x + y \), or monadic-formulas, e.g., \( \text{abs} \ x \). A formula contains at least one operand and at least one operator. The order of elaboration of a formula is determined by the priority of its operators; monadic-formulas are elaborated first and then the dyadic-formulas from the highest to the lowest priority.)

8.4.1. Syntax

a) \* SORTIEVY formula : \* SORTIEVY MOID ADIC formula(b,g,820d,e,f,g).
b) MOID PRIORITY formula(81b,820d,e,f,g,821a,b,822a,b,c,823a,824b,d,825a,b,
c,d,826a,828a) : \* MODE PRIORITY operand(d),
      procedure with \* MODE parameter and \* MODE parameter MOID PRIORITY
      operator(43b), \* MODE PRIORITY plus one operand(d,e).
c) \* operand : \* MODE ADIC operand(d,f).
d) \* MODE PRIORITY operand(b,d) : \* firm MODE PRIORITY formula(820e) ;
      \* MODE PRIORITY plus one operand(d,e).
e) \* MODE priority NINE plus one operand(b,d) : \* MODE monadic operand(f).
f) \* MODE monadic operand(e,g) : \* firm MODE monadic formula(820e) ;
      \* firm MODE secondary(81c).
g) MOID monadic formula(81b,820d,e,f,g,821a,b,822a,b,c,823a,824b,d,825a,b,
c,d,826a,828b) : 
      procedure with \* MODE parameter MOID monadic operator(43c),
      \* MODE monadic operand(f).
h) \* dyadic formula : MOID PRIORITY formula(b).

{Examples:
 b) \( x + x \times y \);
 d) \( x \times y ; x \);
 e) \( x \times y \);
 f) \( \text{abs} \ x ; \text{age of algol} \);
 g) \( \text{abs \ re \ x} \)}

8.4.2. Semantics

A formula is elaborated in the following steps:
Step 1: The formula is replaced by a closed-clause which is a copy of the
routine possessed by the operator-defining occurrence identified by its
operator \( 7.5.2, 4.3.2.b \);
Step 2: The constituent serial-clause of the closed-clause is protected
\( 6.0.2.d \);
Step 3: The skip-symbol \( 5.4.2 \) following the equals-symbol following
its textually first copied formal-parameter is replaced by a copy of the
textually first operand of the formula, and if the formula is a dyadic-
formula, then the skip-symbol following the equals-symbol following its
textually second copied formal-parameter is replaced by a copy of the
textually second operand of the formula;
Step 4: The closed-clause as modified in Steps 2 and 3 is replaced by a
closed-clause consisting of the same sequence of symbols; the elaboration
of this closed-clause is initiated; its value, if any, is then that of the
formula and if this elaboration is completed or terminated, then this
closed-clause is replaced by the formula before the elaboration of a suc-
cessor is initiated.
8.4.2. continued

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

<table>
<thead>
<tr>
<th>dyadic</th>
<th>monadic</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Table" /></td>
<td></td>
</tr>
</tbody>
</table>

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

In the formula $x + y \times 2$, both $y$ and $2$ are primaries, which allows $y$ to be a firm-priority-SEVEN-operand and $z$ to be a firm-priority-EIGHT-operand. The formula $y \times 2$ is then of priority 7. Since $x$ is also a primary, and therefore a firm-priority-SIX-operand; $x + y \times 2$ is a priority-SIX-formula. The effect of $x + y \times 2$ is thus the same as that of $x + (y \times 2)$.

8.5. Cohesions

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

8.5.0.1. Syntax

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

{Examples:

a) real (in sw := real := 3.14) ; re of s

8.5.1. Generators

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

{The elaboration of a generator, e.g., real in sw := real := 3.14 or loc real in ref real x = loc real (usually written real x by extension 9.2.a), involves the creation of a name, i.e., the reservation of storage. The use of a local-generator implies (with most implementations) the reservation of storage in a run-time stack, whereas global-generators imply the
reservation of storage in another region, termed the "heap", in which garbage-collection techniques may be used for storage retrieval. Since this is usually less efficient, global-generators should be avoided where possible. The temptation to use global-generators unnecessarily, is reduced by the extensions 9.2.a, which allow the greatest shortening of the text when local-generators are used.}

8.5.1. Syntax

a) MODE generator(850a) :
   MODE local generator(b,-) ; MODE global generator(c,-).
b) reference to MODE local generator(a) :
   local symbol(31d), actual MODE declarer(71b).
c) reference to MODE global generator(a) :
   heap symbol(31d) option, actual MODE declarer(71b).

{Examples:
a) \texttt{loc real ; heap real ;}
b) \texttt{loc real ;}
c) \texttt{heap real ; real}

8.5.1.2. Semantics

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

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

   (The closed-clause
   \[(\texttt{ref real } xx; xx := (heap real } x := p); x); \texttt{xx = p}\] (see also 9.2.a)
   possesses the value \texttt{true}, but the closed-clause
   \[(\texttt{ref real } xx; xx := (real } x := p); x); \texttt{xx = p}\]
   possesses an undefined value since the name to be assigned to the name pos-
   sessed by xx becomes undefined upon the completion of the elaboration of the
   inner range, which is the scope of the name possessed by x (6.1.2.e, 7.0.2).
   The closed-clause
   \[(\texttt{ref real } xx; \texttt{real } x := p); xx := x) = p;\]
   however, possesses the value \texttt{true}.}

8.5.2. Selections

(A selection selects a field from a structured value; e.g., \texttt{re of } z se-
lects the first real field (usually termed the real part) of the value pos-
sessed by z. If z possesses a name, then \texttt{re of } z possesses also a name, but
if \omega possesses a complex value, then \texttt{re of } \omega possesses a real value, not the
name referring to a real value.)
8.5.2.1. Syntax

a) REPEAT MODE selection(850a): MODE field TAG selector(71j),
   of symbol(3le), weak REPEAT structured with LFIELDSETY MODE field TAG
   LFIELDSETY secondary(81c).

   (Examples, assumed in the reach of the declarations:
   struct language = (int age, ref language fahther);
   language algol := (10, language := (14, "a"); language p11 = (4, algol);
   a) age of p11; father of algol)

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

8.5.2.2. Semantics

A selection is elaborated in the following steps:
Step 1: Its secondary is elaborated; if its value is nil, then the further
elaboration is undefined; otherwise, the structured value which is, or is
referred to by, that value is considered;
Step 2: If the value of the secondary is a name, then the value of the
selection is a new instance of the name which refers to that field of the
considered structured value selected by its field-selector; otherwise, it
is a new instance of {the value which is} that field itself.

(In the examples of 8.5.2.1, age of algol is a reference-to-integral-
selection, and, by 8.5.0.1.a, a reference-to-integral-cohesion, but age of
p11 is an integral-selection and an integral-cohesion. It follows that age
of algol may appear as a destination (8.3.1.1.b) in an assignation, but age
of p11 may not. Similarly, algol is a reference-to-[language]-base, but p11
is a [language]-base and no assignation may be made to p11. (Here, [language]
stands for structured-with-integral-field-[age]-and reference-to-[language]-
field-[father] and [age] stands for letter-a-letter-g-letter-e, etc.) The
selection father of p11, however, is a reference-to-[language]-selection and
thus a reference-to-[language]-cohesion whose value is the name possessed by
algol. It follows that the identity-relation father of p11 := algol possess-
es the value true. If father of p11 is used as a destination in an assigna-
tion, then there is no change in the name which is a field of the structured
value possessed by p11, but there may well be a change in the value of mode
[language] referred to by that name. By similar reasoning and because the
operators re and im possess routines (10.2.7.b,c) which deliver values whose
mode is 'real' and not 'reference to real', re of z := im w is an assigna-
tion, but re z := im w is not.)
8.6. Bases

{Bases are mode-identifiers, e.g., x, denotations, e.g., 3.14, slices, e.g., x1[ i ] and calls, e.g., sin(x). Bases are, generally, elaborated first. They may be subscripted, parametrized and selected from and are often used as operands. Moreover, certain void-bases are void-cast-packs, e.g., (: x := x + 1), which may be used, e.g., as procedure-cocorends; it is essential that they begin with an open-symbol and end with a close-symbol for, otherwise, the parsing of, e.g., a := b, which might, in practice, be indistinguishable from a :=: b, would depend on the modes of a and b.}

8.6.0.1. Syntax

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

b)  \text{void base(81d,820d,e,f,g,823a) ; void call(862a) ; void cast(834a) pack.}

(Examples:
  a)  \text{\( x ; 3.14 ; x2[ i, j ] ; \sin(x) ; \)}
  b)  \text{\( \text{Look (stand in)} ; (: x := 3.14) \)}

8.6.0.2. Semantics

a)  A mode-identifier is elaborated by making a copy of the instance of the value, if any, possessed by the defining occurrence identified by it \{4.1.2, 7.4.2.Step 7\}; its value is the copy.

b)  The elaboration of a void-cast-pack is that of its void-cast.

8.6.1. Slices

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

8.6.1.1. Syntax

a)  \text{\texttt{REPEITY ROWSEITY NONROW slice(860a) :}}
    \text{\texttt{weak REPEITY ROWS ROWSEITY NONROW primary(81d), sub symbol(31e),}}
    \text{\texttt{ROWS leaving ROWSEITY indexer(b,c,d,e,-), bus symbol(31e).}}

b)  \text{\texttt{row of ROWS leaving row of ROWSEITY indexer(a,b) :}}
    \text{\texttt{trimmer(i), comma symbol(31e),}}
    \text{\texttt{ROWS leaving ROWSEITY indexer(b,c,d,e,-) ;}}
    \text{\texttt{subscript(i), comma symbol(31e),}}
    \text{\texttt{ROWS leaving row of ROWSEITY indexer(b,d,-).}}

c)  \text{\texttt{row of ROWS leaving EMPTY indexer(a,b,c) :}}
    \text{\texttt{subscript(i), comma symbol(31e), ROWS leaving EMPTY indexer(c,e).}}

d)  \text{\texttt{row of leaving row of indexer(a,b) : trimmer(i).}}

e)  \text{\texttt{row of leaving EMPTY indexer(a,b,c) : subscript(i).}}
8.6.1.1. continued

f) \textit{trimmer}(b,a) : strict lower bound(71u) option, up to symbol(31e), strict upper bound(71u) option, new lower bound part(g) option.
g) new lower bound part(f) : at symbol(31e), new lower bound(h).
h) new lower bound(g) : strong integral tertiary(81b).
i) \textit{subscript}(b,c,e) : strong integral tertiary(81b).
j) \textit{trimscript} : trimmer(f) ; subscript(i).
k) \textit{indexer} : ROWS leaving ROWSETY indexer(b,c,d,e).
l) \textit{boundscript} : strict LOWER bound(71u) ; new lower bound(h) ; subscript(i).

\{Examples:
  a) z1 [i] ; z2 [i, j] ; z2 [i] ; z1 [2 : n] ;
  b) z : n, j ; l, 2 : n ;
  c) \textit{i}, \textit{j} ;
  d) \textit{z} ;
  e) \textit{i} ;
  f) z : n ; z : n \& \textit{z} ;
  g) \textit{z} ;
  h) 0 ;
  i) \textit{i} ;
\}

\{In rule a, 'ROWS' reflects the number of trimscripts in the slice, 'ROWSETY' the number of these which are trimmers and 'ROWSETY' the number of 'row of' not involved in the indexer. In the slices z2 [i, j], z2 [i, 2 : n], z2 [i], these numbers are \{2, 0, 0\}, \{2, 1, 0\} and \{1, 0, 1\} respectively. Because of rules f and 7.1.1.u, 2 : n \& 0, 2 : n, 2 ::, :: 5 and :: 0 are trimmers.\}

8.6.1.2. Semantics

A slice is elaborated in the following steps:
Step 1: Its primary, and all its constituent boundscripts not contained in its primary, are elaborated collaterally (6.2.2.a); if the value of the primary is \textit{nil}, then the further elaboration is undefined; otherwise, Step 2 is taken;
Step 2: The multiple value which is, or is referred to by, the value of the primary, is considered, a copy is made of its descriptor, and all the states (2.2.3.3.b) in the copy are set to 1;
Step 3: The trimscript following the sub-symbol is considered and a pointer, \textit{\iota}, is set to 1;
Step 4: If the considered trimscript is an up-to-symbol, then Step 6 is taken; otherwise, if it is a trimmer, then Step 5 is taken; otherwise, letting \textit{k} stand for its value, if \textit{\iota} \leq \textit{k} \leq \textit{u}, then the offset in the copy is increased by \( (k - \textit{\iota}) \times \textit{d} \), the \textit{\iota}-th quintuple is "marked", and Step 6 is taken; otherwise, the further elaboration is undefined;
Step 5: The values \textit{\ell}, \textit{u} and \textit{\ell}' are determined from the considered trimscript as follows:
  if the considered trimscript contains a strict-lower-bound (strict-upper-bound), then \textit{\ell} (\textit{u}) is its value; otherwise, \textit{\ell} (\textit{u}) is \textit{\ell} (\textit{u}); if it contains a new-lower-bound, then \textit{\ell}' is its value; otherwise, \textit{\ell}' is 1;
  if now \ell < \ell and \textit{u} \leq \textit{u}, then the offset in the copy is increased by \( (\textit{\ell} - \textit{\ell}) \times \textit{d} \), and then \textit{\ell} is replaced by \textit{\ell}' and \textit{u} by \( (\textit{\ell}' - \textit{\ell}) + \textit{u} ;
  otherwise, the further elaboration is undefined;
8.6.1.2. continued

Step 6: If the considered transcript is followed by a comma-symbol, then the transcript 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 obtained by deleting the initial 'reference to', if any, from the mode enveloped by the original of the slice, is considered instead; otherwise, the element of the considered value selected by (the index equal to) the offset in the copy is considered instead;
Step 8: If the value of the primary is a name, then the value of the slice is a new instance of the name which refers to the considered value, and, otherwise, is a new instance of the considered value itself.

(A trimmer restricts the possible values of a subscript and changes its notation: first, the value of the subscript is restricted to run from the value of the strict-lower-bound to the value of the strict-upper-bound, both given in the old notation; next, all restricted values of that subscript are changed by adding the same amount to each of them, such that the lowest value then equals the value of the new-lower-bound. Thus, the assignments \( y1 \{ 1 : n - 1 \} := x1 \{ 2 : n \}; y1 \{ n \} := x1 \{ 1 \}; x1 := y1 \) effect a cyclic permutation of the elements of \( x1 \).)

8.6.2. Calls

(Calls are obtained by parametrizing, e.g., \( \sin (x + 1) \). Parametrizing may be done only to primaries, e.g., \( \sin \) and \( \cos \) or (\( p \mid \sin \mid \cos \)). The completed elaboration of a call may or may not deliver a value.)

8.6.2.1. Syntax

a) MOID call(b60s,b) : firm procedure with PARAMETERS MOID primary(81d),
actual PARAMETERS(54c,74b) pack.

{Example:
a) \( \sin (x) \) }

8.6.2.2. Semantics

A call is elaborated in the following steps:
Step 1: Its primary is elaborated;
Step 2: The call is replaced by a closed-clause which is a copy of (the routine which is) the value obtained in Step 1;
Step 3: The constituent serial-clause of the closed-clause is protected {6.0.2.d};
Step 4: The skip-symbols (5.4.2. Step 2) following the equals-symbols following the copied formal-parameters are replaced in the textual order by copies of the constituent actual-parameters of the call taken in the same order;
8.6.2.2. continued

Step 5: The closed-clause as modified in Steps 3 and 4 is replaced by a closed-clause consisting of the same sequence of symbols; the elaboration of this closed-clause is initiated; its value, if any, is then that of the call and if this elaboration is completed or terminated, then this closed-clause is replaced by the call before the elaboration of a successor is initiated.

(The call \texttt{samealon (m, (int j) real : x1 [j])} in the reach of the declaration

\begin{verbatim}
proc samealon = (int n, proc (int) real f) real :
  begin long real s := long 0; for i to n do s += length f (i) + 2;
  short long sqrt (s)
end
\end{verbatim}

is elaborated by replacing it (Step 2) by the closed-clause

\begin{verbatim}
(int n = m, proc (int) real f = (int j) real : x1 [j]); real :
  begin long real s := long 0; for i to n do s += length f (i) + 2;
  short long sqrt (s)
end
\end{verbatim}

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

\begin{verbatim}
(int n = m, proc (int) real f = (int j) real : x1 [j]); real :
  begin long real s := long 0; for i to n do s += length f (i) + 2;
  short long sqrt (s)
end
\end{verbatim}

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

a) An extension is the insertion of a comment between two symbols or the replacement of a certain sequence of symbols, possibly satisfying certain restrictions, by another sequence of symbols, as indicated in sections 9.1 up to 9.4.

b) No extension may be performed within a comment \{3.0.9.b\}, character-denotation \{5.1.4.1.a\}, or row-of-character-denotation \{5.3.1.b\}.

c) Some extensions are given in the representation language, except that
   A, B and C stand for strong-unitary-integral-clauses \{8.1.1.a\},
   D for a strong-serial-boolean-clause \{6.1.1.a\},
   E for a strong-unitary-vold-clause \{8.1.1.a\},
   F and G for unitary-clauses \{8.1.1.a\},
   H for two or more unitary-clauses \{8.1.1.a\} separated by comma-symbols \{3.1.1.3\},
   I, J, K and L for mode-identifiers \{4.1.1.b\},
   M for a label-identifier \{4.1.1.b\},
   N for a local-symbol \{3.1.1.a\} or heap-symbol \{3.1.1.d\}-option,
   O for a conformity-relator \{8.3.2.1.b\},
   P for an indication \{4.2.1.a\},
   Q for a virtual-plan \{7.1.1.x.a\},
   R for a routine-denotation \{5.4.1.a\},
   S for the standard- Prelude \{2.1.b, 10\} if the extension is performed outside
   the standard-prelude and, otherwise, for the empty sequence of symbols,
   T for a condition followed by a choice-clause \{6.4.1.b,c,d\},
   U for a declarer \{7.1.1.a\},
   V for a virtual-declarer \{7.1.1.b\} or for a formal-declarer \{7.1.1.b\} all of
   whose constituent formal-lower-bounds and formal-upper-bounds are either-
   symbols,
   W, X and Y for tertiaries \{8.1.1.b\},
   Z for two or more tertiaries \{8.1.1.b\} separated by comma-symbols \{3.1.1.e\},
   \(\Gamma\) for a comma-symbol \{3.1.1.e\}, go-on-symbol \{3.1.1.f\} or becomes-symbol
   \{3.1.1.c\},
   \(\Sigma\) for a serial-clause \{6.1.1.a\},
   \(\Phi\) for a VICTAL-ROWS-rower \{7.1.1.q.r\} or indexer \{8.6.1.1.k\}, where "VICTAL"
   ("ROWS") stands for any terminal production of the metanotion 'VICTAL'
   ('ROWS'),
   \(\Delta\) for an open-symbol \{3.1.1.e\}, and
   \(\Psi\) for a close-symbol \{3.1.1.e\}.

d) Each representation of a symbol appearing in sections 9.1 up to 9.4 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
9.1).

\{(e.g., (m > n | m | n)) may be replaced by
(m > n | m \(\Phi\) the larger of the two \(\Phi\) | n).\}
9.2. Contractions

a) \( \text{ref} \; V \; I = \text{loc} \; U \; I \) and \( \text{ref} \; V \; I = \text{heap} \; U \; I \) where \( \text{ref} \; V \; I \) is the formalparameter of an identity-declaration (7.4.1.a) which is not followed by a comma-symbol (3.1.1.e) followed by a mode-identifier (4.1.1.b) and where \( U \) and \( V \) specify the same mode (7.1.2.a) may be replaced by \( U \; I \; \Gamma \) and \( \text{heap} \; U \; I \; \Gamma \) respectively.

(e.g., \( \text{ref} \; \text{real} \; x = \text{loc} \; \text{real}; \) may be replaced by \( \text{real} \; x; \), \( \text{ref} \; \text{bool} \; p = \text{loc} \; \text{bool} : = \text{true} \) may be replaced by \( \text{bool} \; p : = \text{true} \), and \( \text{ref} \; \text{real} \; t = \text{heap} \; \text{real} ; \) may be replaced by \( \text{heap} \; \text{real} ; \).

b) \( \text{mode} \; P = \text{struct} \) may be replaced by \( \text{struct} \; P = \) and \( \text{mode} \; P = \text{union} \) by \( \text{union} \; P = \).

(e.g., \( \text{mode} \; \text{compl} = \text{struct} \; \text{real} \; \text{re}, \; \text{im} \)) (see also 9.2.c) may be replaced by \( \text{struct} \; \text{compl} = (\text{real} \; \text{re}, \; \text{im}) \).

c) If a given mode-declaration (7.2.1.a) (priority-declaration (7.3.1.a), identity-declaration (7.4.1.a), operation-declaration (7.5.1.a), formalparameter (5.4.1.e), field-declarator (7.1.1.g)) and another one following a comma-symbol (3.1.1.e) following the given one both begin with a mode-symbol, structure-symbol, union-of-symbol, priority-symbol, operation-symbol (all 3.1.1.d), or one same terminal production of "VICTUAL MODE declamer" (7.1.1.b) or of "MODE global generator" (8.5.1.a) where "MODE" ("VICTUAL") stands for any terminal production of the metaction "MODE" ("VICTUAL"), then the second of these occurrences may be omitted.

(e.g., \( \text{real} \; x, \; \text{real} \; y :: = 1.2 \) may be replaced by \( \text{real} \; x, \; y :: = 1.2 \), but \( \text{real} \; x, \; \text{real} \; y :: = 1.2 \) may not be replaced by \( \text{real} \; x, \; y :: = 1.2 \), since the first occurrence of \( \text{real} \) is an actual-declarer whereas the second is a formal-declarer. Note also that \( \text{mode} \; b = \text{bool}, \; \text{mode} \; r = \text{real} \) may be replaced by \( \text{mode} \; b = \text{bool}, \; r = \text{real} \), etc.)

d) If an actual-parameter (7.4.1.b) (source (8.3.1.1.c)) is a routine-denotation (5.4.1.a) or a void-cast-pack (8.3.4.1.a) (is a routine-denotation), then its first open-symbol and last close-symbol (both 3.1.1.e) may simultaneouly be omitted.

(e.g., \( \text{op} \; + = (\text{int} \; a) \; \text{int} :: = a \) may be replaced by \( \text{op} \; + = (\text{int} \; a) \; \text{int} : = a \).)

e) If the original (1.1.6.c) of \( Q \) and the original of \( R \) envelop (1.1.6.j) the same mode, then the unitary-phrase \( \text{proc} \; Q \; I = R \) \( \text{op} \; Q \; P = R \), \( N \; \text{proc} \; Q = R \) may be replaced by \( \text{proc} \; I = R \) \( \text{by} \; P \; Q = R \), \( N \; \text{proc} : = R \). (e.g., \( \text{proc} \; (\text{ref} \; \text{int}) \; \text{inor} = (\text{ref} \; \text{int}) \; i :: = 1 \) may be replaced by \( \text{proc} \; \text{inor} = (\text{ref} \; \text{int}) \; i :: = 1, \text{op} \; (\text{ref} \; \text{int}) \; \text{deor} = (\text{ref} \; \text{int}) \; i \) \( \text{int} :: = 1 \) may be replaced by \( \text{op} \; \text{deor} = (\text{ref} \; \text{int}) \; i \) \( \text{int} :: = 1 \), and the actual-parameter of the identity-declaration \( \text{ref} \; \text{proc} \; (\text{real}) \; \text{int} \; v = \text{loc} \; \text{proc} := ((\text{real} \; x) \; \text{int} :: = \text{round} \; x) \) may be replaced by \( \text{loc} \; \text{proc} : = ((\text{real} \; x) \; \text{int} :: = \text{round} \; x) \), whereupon application of 9.2.a,d may yield the identity-declaration \( \text{proc} \; p :: = (\text{real} \; x) \; \text{int} :: = \text{round} \; x) \).

f) \[ : \] may be replaced by \[ : \], \[ : \] may be replaced by \[ : \], \[ : \] may be replaced by \[ : \], \[ : \] may be replaced by \[ : \], \[ : \] may be replaced by \[ : \].

(e.g., \[ : \] \( \text{real} \) may be replaced by \[ : \] \( \text{real} \).)

g) \[ : \] may be replaced by \( \Delta \) or by \( \Delta / \).

(e.g., \[ : \] \( \text{int} \) may be replaced by \( \text{int} \) or by \( \text{int} \).)
9.3. Repetitive statements

a) The strong-unitary-void-clause (8.1.1.a)

\begin{verbatim}
begin int J := A, int K := B, L := C;
M: if S (K > 0 | \(J \leq L\) | K < 0 | J \(\geq L\) | true)
then int I := J; (D | E; (S J :=:= K); go to M)
end
\end{verbatim}

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

\begin{verbatim}
for I from A by B to C while D do E
\end{verbatim}

and if, moreover, \(I\) does not occur in \(D\) or \(E\), then \textit{for I from} may be replaced by \textit{from}. 

b) The strong-unitary-void-clause (8.1.1.a)

\begin{verbatim}
begin int J := A, int K := B;
M: (int I := J; (D | E; (S J :=:= K); go to M))
end
\end{verbatim}

where \(J, K\) and \(M\) do not occur in \(D, E\) or \(S\), and where \(I\) differs from \(J\) and \(K\), may be replaced by

\begin{verbatim}
for I from A by B while D do E
\end{verbatim}

and if, moreover, \(I\) does not occur in \(D\) or \(E\), then \textit{for I from} may be replaced by \textit{from}.

c) \textit{from I by} may be replaced by \textit{by}, \textit{by I to} by \textit{to}, \textit{by I while} by \textit{while}, and \textit{while true do} by \textit{do}.

\begin{verbatim}
\{e.g., for i from 1 by 1 to n while true do x ::= x1[i]\} may be replaced by
\{for i to n do x ::= x1[i]\}. Note that to \(0\) do \(E\) and while false do \(E\) do not cause \(E\) to be elaborated at all, whereas do \(E\) causes \(E\) to be elaborated repeatedly until the elaboration is terminated, interrupted or halted.\}

9.4. Contracted conditional clauses

\begin{verbatim}
\{The flowers that bloom in the spring,
\textit{Tra la},
Have nothing to do with the case.
\textit{Mikado,}
\textit{W.S. Gilbert.}\}
\end{verbatim}

a) \textit{else skip fi} may be replaced by \textit{fi}.
\begin{verbatim}
\{e.g., if x < 0 then z ::= 0 else skip fi may be replaced by if x < 0 then z ::= 0 fi.\}
\end{verbatim}

b) \textit{else if T fi fi} may be replaced by \textit{elsif T fi} and
\begin{verbatim}
\{then if T fi fi may be replaced by that T fi.\}
\end{verbatim}

\begin{verbatim}
\{e.g., if P then princeton else if Q then grenoble else sandvoort fi fi may be replaced by if P then princeton elsif Q then grenoble elsif sandvoort fi fi or by (p | princeton ; q | grenoble | sandvoort). Many more examples are to be found in 10.5.\}
\end{verbatim}

c) \textit{(int I := A; if S I = 1 then F else S I = 3 then G else \Sigma fi), where I does not occur in F, G, S or \Sigma, may be replaced by case \(A\ in F, G\ out \Sigma\ esac (or by (A | F, G | \Sigma)).}\}

d) \textit{(int I := A; if S I = 1 then F else case (S I - 1) in H out \Sigma\ esac fi), where I does not occur in F, H, S or \Sigma, may be replaced by case \(A\ in F, H\ out \Sigma\ esac (or by (A | F, H | \Sigma)).}\}

\{Examples of the use of such "case clauses" are given in 11.11.v,am and 11.12.\}
9.4. continued

e) \begin{align*}
& (\text{int } I, \text{ sema } L = (S / 1); V K = W; \\
& \text{par } ((X O K | S + L; I := I; M), \\
& \quad (Y O K | S + L; I := 2; M)); 0. M; I),
\end{align*}

where \( W \) is the same as some terminal production of 'MODE tertiary' in which
"MODE" stands for the mode specified by \( V \), and where \( I, K, L \) and \( M \) do not
occur in \( W, X \) and \( Y \), may be replaced by \([*X, Y O W*]\).

g) \begin{align*}
& (\text{int } I, J; \text{ sema } L = (S / 1); V K = W; \\
& \text{par } ((X O K | S + L; I := I; M), \\
& \quad (S (J := ((Z O W*) + 1)) > I | S + L; I := J; M)); 0. M; I),
\end{align*}

where \( W \) is the same as some terminal production of 'MODE tertiary' in which
"MODE" stands for the mode specified by \( V \), and where \( I, J, K, L \) and \( M \) do not
occur in \( S, W, X \) or \( Z \), may be replaced by \([*X, Z O W*]\).

([*Z O W*]) \mid R \mid \Sigma) may be replaced by \textbf{case 2 O W in H out} \Sigma \textbf{ esac}.

(Examples of the use of such "conformity case clauses" are given in
11.11.q,ah.)
10. Standard prelude and postlude

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

b) A representation of the standard-prelude is obtained by altering each form in 10.1, 10.2, 10.3, 10.4 and 10.5 in the following steps:
Step 1: Each sequence of symbols between $\#$ and $\&$ in a given form is altered in the following steps:
Step 1.1: If $D$ occurs in the given sequence of symbols, then the given sequence is replaced by a chain of a sufficient number of sequences separated by comma-symbols; the first new sequence is a copy of the given sequence in which copy $D$ is deleted; the $n$-th new sequence, $n > 1$, is a copy of the given sequence in which copy $D$ is replaced by a sub-symbol followed by $n - 1$ comma-symbols followed by a bus-symbol;
Step 1.2: If, in the given sequence of symbols, as possibly modified in Step 1.1, $L \text{ int}$, $L \text{ real}$ or $L \text{ compl}$ occurs, then that sequence is replaced by a chain of a sufficient number of 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 $L$ ($L$);
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, $L \text{ int}$ ($L \text{ real}$, $L \text{ compl}$, $L \text{ byte}$, $L \text{ bytes}$) occurs, then the form is replaced by a sequence of a sufficient number of new forms; the $n$-th new form is a copy of the given form in which copy each occurrence of $L$ ($L$, $L$) is replaced by $(n - 1)$ times $L$ ($L$, $L$, 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 $\times$ or $/$;
Step 5: If $Q$ occurs in a given form, as possibly modified or made in the Steps above, then the form is replaced by four new forms obtained by replacing $Q$ consistently throughout the form by either $-=$ or $+=+$ or $\times=$ or $/=$;
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 $\neq$ or $\geq$ or $\leq$;
Step 7: If $E$ occurs in a given form, as possibly modified or made in the Steps above, then the form is replaced by two new forms obtained by replacing $E$ consistently throughout the form by either $=$ or $\neq$;
Step 8: Each occurrence of $F$ in any form, as possibly modified or made in the Steps above, is replaced by a representation of 'letter aleph symbol' ($5.5.8$);
Step 9: If, in some form, as possibly modified or made in the Steps above, $\%$ occurs followed by the representation of an identifier (field-selector, induction), then that occurrence of $\%$ is deleted and each occurrence of that representation in any form is replaced by one same representation of an identifier (field-selector, induction) which does not occur elsewhere in the program and Step 9 is taken;
10. continued

Step 10: If a sequence of representations beginning with and ending with \( a \) occurs in any form, as possibly modified or made in the steps above, then this sequence is replaced by a representation of an actual-declarer or closed-clause suggested by the sequence;

Step 11: If, in any form, as possibly modified or made in the steps above, a representation of a routine-denotation occurs whose elaboration involves the manipulation of real numbers, then this denotation may be replaced by any other denotation whose elaboration has approximately the same effect (the degree of approximation is left undefined in this Report (see also 2.2.3.1.c)).

Step 12: The standard-prelude is that declaration-prelude-sequence 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.)

c) A representation of the standard-postlude is given in 10.6.

10.1. Environment enquiries

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

b) \( L \) int \( L \) max int = a the largest \( L \) integral value a;

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

d) \( L \) real \( L \) max real = a the largest \( L \) real value a;

e) \( L \) real \( L \) small real = a the smallest \( L \) real value such that both
\[ L 1 + L \text{ small real } > L 1 \text{ and } L 1 - L \text{ small real } < L 1 a; \]

f) int bits widths = a the number of different widths of bits a;

g) int i bits width =
\[ a \text{ the number of elements in } L \text{ bits}; \text{ see } L \text{ bits } (10.2.3.a) a; \]

h) int bytes widths = a the number of different widths of bytes a;

i) int i bytes width =
\[ a \text{ the number of elements in } L \text{ bytes}; \text{ see } L \text{ bytes } (10.2.4.a) a; \]

j) op abs = (char a) int :
\[ a \text{ the integral equivalent of the character } 'a' a; \]

k) op repr = (int a) char :
\[ a \text{ that character } 'a', \text{ if it exists, for which abs } x = a a; \]

1) char null character = a some character a;

10.2. Standard priorities and operations

10.2.0. Standard priorities

a) priority = := = 1, += = 1, *= = 1, /= = 1, =:= = 1, +=: = 1, 
\[ v = 8, \land = 8, = = 8, \neq = 8, \leq = 8, \geq = 8, > = 8, < = 6, += = 6, 
\[ * = 7, : = 7, =: = 7, /= = 7, [] = 8, = 8, \land = 8, \lor = 8, \lnot = 8, \]
10.2.1. Rows and associated operations

a) \( \text{mode} \% \text{rows} = \text{a} \) an actual-declarer specifying a mode united from (4, 3, a) all modes beginning with 'row of' \( \text{a} \);

b) \( \text{op} \; L = (\text{int} \; n, \text{rows} \; a) \; \text{int} : \text{a} \) the lower bound of the \( n \)-th quintuple of the descriptor of the value of 'a', if that quintuple exists \( \text{a} \);

c) \( \text{op} \; U = (\text{int} \; n, \text{rows} \; a) \; \text{int} : \text{a} \) the upper bound of the \( n \)-th quintuple of the descriptor of the value of 'a', if that quintuple exists \( \text{a} \);

d) \( \text{op} \; ( = (\text{int} \; n, \text{rows} \; a) \; \text{bool} : \text{a} \) true \) (false) if the lower state in the \( n \)-th quintuple of the descriptor of the value of 'a' equals 1 (0), if that quintuple exists \( \text{a} \);

e) \( \text{op} \; ( = (\text{int} \; n, \text{rows} \; a) \; \text{bool} : \text{a} \) true \) (false) if the upper state in the \( n \)-th quintuple of the descriptor of the value of 'a' equals 1 (0), if that quintuple exists \( \text{a} \);

f) \( \text{op} \; L = (\text{rows} \; a) \; \text{int} : 1 \; \text{a} ;

g) \( \text{op} \; U = (\text{rows} \; a) \; \text{int} : 1 \; \text{a} ;

h) \( \text{op} \; ( = (\text{rows} \; a) \; \text{bool} : 1 \; \text{a} ;

i) \( \text{op} \; ( = (\text{rows} \; a) \; \text{bool} : 1 \; \text{a} ;

10.2.2. Operations on boolean operands

a) \( \text{op} \; v = (\text{bool} \; a, b) \; \text{bool} : (a \; \text{true} \; b) \);

b) \( \text{op} \; \land = (\text{bool} \; a, b) \; \text{bool} : (a \; b \; \text{false}) \);

c) \( \text{op} \; \neg = (\text{bool} \; a) \; \text{bool} : (a \; \text{false} \; \text{true}) \);

d) \( \text{op} \; = = (\text{bool} \; a, b) \; \text{bool} : (a \; b) \; v (\neg a \; \neg b) \);

e) \( \text{op} \; \neq = (\text{bool} \; a, b) \; \text{bool} : (a \; b) \);

f) \( \text{op} \; \text{abs} = (\text{bool} \; a) \; \text{int} : (a \; 1 \; 0) ;

10.2.3. Operations on integral operands

a) \( \text{op} \; < = (\text{L int} \; a, b) \; \text{bool} : \text{a}\) true if the value of 'a' is smaller than \( \{2.2.3.1.0\} \) that of 'b' and false otherwise \( \text{a} \);

b) \( \text{op} \; \leq = (\text{L int} \; a, b) \; \text{bool} : \neg (b < a) ;

c) \( \text{op} \; \geq = (\text{L int} \; a, b) \; \text{bool} : a \leq b \; \text{b} \leq a ;

d) \( \text{op} \; \neq = (\text{L int} \; a, b) \; \text{bool} : \neg (a = b) ;

e) \( \text{op} \; = = (\text{L int} \; a, b) \; \text{bool} : b \leq a ;

f) \( \text{op} \; > = (\text{L int} \; a, b) \; \text{bool} : b < a ;

g) \( \text{op} \; = = (\text{L int} \; a, b) \; \text{L int} : a \; b \; \text{a value of 'a' minus \{2.2.3.1.0\} that of 'b', 'a'} ;

h) \( \text{op} \; - = (\text{L int} \; a) \; \text{L int} : \text{L} \; 0 \; a ;

i) \( \text{op} \; + = (\text{L int} \; a, b) \; \text{L int} : a \; b ;

j) \( \text{op} \; += (\text{L int} \; a) \; \text{L int} : a ;

k) \( \text{op} \; \text{abs} = (\text{L int} \; a) \; \text{L int} : (a < \text{L} \; 0 \; a \; \text{a} \; a) ;

l) \( \text{op} \; \times = (\text{L int} \; a, b) \; \text{L int} : (\text{L int} \; s := \text{L} \; 0, i := \text{abs} b) ;

\text{while} \; i \geq \text{L} \; 1 \; \text{do} \; (s := a + a; i := i - \text{L} \; 1) ; (b < \text{L} \; 0 \; 1 \; s) ; (a < \text{L} \; 0 \; b \leq \text{L} \; 0 \; 0 \; a = \text{L} \; 0 \; b < \text{L} \; 0 \; 0 \; a = \text{L} \; 0 \; q) ;

m) \( \text{op} \; i = (\text{L int} \; a, b) \; \text{L int} : (b \; \text{L} \; 0 \; \text{L int q} := \text{L} \; 0, r := \text{abs} a ;

\text{while} \; (r := r - \text{abs} b) = \text{L} \; 0 \; \text{do q} := q + \text{L} \; 1 ;

(a < \text{L} \; 0 \; b \geq \text{L} \; 0 \; 0 \; a \leq \text{L} \; 0 \; b < \text{L} \; 0 \; 1 = \text{q} = \text{q}) ;

n) \( \text{op} \; += (\text{L int} \; a, b) \; \text{L int} : (r = a - a \; b \times b ; (r < b) r \; \text{abs} b ; \text{r}) ;

o) \( \text{op} \; / = (\text{L int} \; a, b) \; \text{L real} : (\text{L real} \; a) / (\text{L real} \; b) ;

p) \( \text{op} \; += (\text{L int} \; a, b) \; \text{L int} : (b \; 0 \; \text{L int p} := \text{L} \; 1, \text{to b do p} := p \times a ; p) ;

q) \( \text{op} \; \text{len} = (\text{L int} \; a) \; \text{L int} : \text{a} \) the long L integral value equivalent to \{2.2.3.1.0\} the value of 'a' 'a' ;
10.2.3. continued

r) \text{op short} = (\text{long } L \text{ int } a) \ L \text{ int} : c \text{ the } L \text{ integral value, if it exists, equivalent to } (2.2.3.1.d) \text{ the value of } 'a' / c ;

s) \text{op odd} = (L \text{ int } a) \ \text{bool} : \text{abs } a : L 2 = L 1 ;

t) \text{op sign} = (L \text{ int } a) \ \text{int} : (a > L 0 | 1 | : a < L 0 | - 1 | 0) ;

u) \text{op i} = (L \text{ int } a, b) \ L \text{ compl} : (a, b) ;

10.2.4. Operations on real operands

a) \text{op } < = (L \text{ real } a, b) \ \text{bool} : c \text{ true if the value of } 'a' \text{ is smaller than } (2.2.3.1.c) \text{ that of } 'b' \text{ and false otherwise } a ;

b) \text{op } \leq = (L \text{ real } a, b) \ \text{bool} : \neg (b < a) ;

c) \text{op } = = (L \text{ real } a, b) \ \text{bool} : a \leq b \land b \leq a ;

d) \text{op } + = (L \text{ real } a, b) \ \text{bool} : \neg (a = b) ;

e) \text{op } < = (L \text{ real } a, b) \ \text{bool} : b \leq a ;

f) \text{op } = = (L \text{ real } a, b) \ \text{bool} : b < a ;

g) \text{op } - = (L \text{ real } a, b) \ L \text{ real} : c \text{ the value of } 'a' \text{ minus } (2.2.3.1.c) \text{ that of } 'b' / c ;

h) \text{op } / = (L \text{ real } a, b) \ L \text{ real} : c \text{ the value of } 'a' \text{ divided by } (2.2.3.1.c) \text{ that of } 'b' / c i ;

i) \text{op } \text{leng} = (L \text{ real } a) \ L \text{ real} : c \text{ the long } L \text{ real value equivalent to } (2.2.3.1.d) \text{ the value of } 'a' / c ;

j) \text{op short} = (\text{long } L \text{ real } a) \ L \text{ real} : c \text{ if } \text{abs } a \leq \text{ leng } L \text{ max real, then } a \text{ L real value } 'v' \text{ such that, for any } L \text{ real value } 'w', \text{abs } (\text{leng } v - a) \leq \text{abs } (\text{leng } w - a) / c ;

k) \text{op round} = (L \text{ real } a) \ L \text{ int} : c \text{ a } L \text{ integral value, if one exists, equivalent to } (2.2.3.1.a) \text{ a } L \text{ real value differing by not more than one-half from the value of } 'a' / c ;

l) \text{op sign} = (L \text{ real } a) \ \text{int} : (a > L 0 | 1 | : a < L 0 | - 1 | 0) ;

m) \text{op enter} = (L \text{ real } a) \ \text{int} : (L \text{ int } j := L 0 ; \text{while } j < a \ \text{do } j := j + L 1 ; \text{while } j < a \ \text{do } j := j - L 1 ; j) ;

n) \text{op i} = (L \text{ real } a, b) \ L \text{ compl} : (a, b) ;

10.2.5. Operations on arithmetic operands

a) \text{op } F = (L \text{ real } a, L \text{ int } b) \ L \text{ real} : a F (L \text{ real } b) ;

b) \text{op } P = (L \text{ int } a, L \text{ real } b) \ L \text{ real} : (L \text{ real } a) F b ;

c) \text{op } R = (L \text{ real } a, L \text{ int } b) \ \text{bool} : a R (L \text{ real } b) ;

d) \text{op } F = (L \text{ int } a, L \text{ real } b) \ \text{bool} : (L \text{ real } a) F b ;

e) \text{op i} = (L \text{ real } a, L \text{ int } b) \ \text{compl} : (a, b) ;

f) \text{op i} = (L \text{ int } a, L \text{ real } b) \ \text{compl} : (a, b) ;

g) \text{op i} = (L \text{ real } a, L \text{ real } b) \ L \text{ real} : (L \text{ real } p := L 1 ; \text{to abs } b \ \text{do } p := p = a ; (b \geq 0 | p | \ L 1 / p)) ;

10.2.6. Operations on character operands

a) \text{op } R = (\text{char } a, b) \ \text{bool} : \text{abs } a R \text{ abs } b ; \{10.1.j\}

b) \text{op } F = (\text{char } a, b) \ \text{string} : (a, b) ;
10.2.7. 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 : 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 \land im a = im b;  
g) op * = (L compl a, b) L compl : ~ (a = b);  
h) op - = (L compl a, b) L compl : (re a - re b) \land (im a - im b);  
i) op * = (L compl a, b) L compl : (re a + re b) \land (im a + im b);  
j) op + = (L compl a, b) L compl : (re a - re b) \land (im a - im b);  
k) op * = (L compl a) L compl : a;  
l) op * = (L compl a, b) L compl : 
   (re a \times re b - im a \times im b) \land (re a \times im b + im a \times re b);  
m) op / = (L compl a, b) L compl : (L real d = re (b \times conj b));  
   L compl n = a \times conj b; (re n / d) \land (im n / d);  

n) op leng = (L compl a) L compl : leng re a \land leng im a;  

o) op short = (long L compl a) L compl : short re a \land 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;  

10.2.8. Bits structures and associated operations

a) struct L bits = (L int, bits width) bool L F); {See 10.1.9}  
   (The field-selector is hidden from the user in order that he may not 
   break open the structure; in particular, he may not subscript the field.)  
b) op = = (L bits a, b) bool : (for i to L bits width do 
   ((L F of a) [i] = (L F of b) [i]) \land true; L : false);  
c) op # = (L bits a, b) bool : ~(a = b);  
d) op v = (L bits a, b) L bits : (L bits a; for i to L bits width do 
   (L F of a) [i] \land (L F of b) [i]); a);  
e) op * = (L bits a, b) L bits : (L bits a; for i to L bits width do 
   (L F of a) [i] \land (L F of b) [i]); a);  
f) op <= = (L bits a, b) bool : (a = b) \lor (b < a);  
g) op >= = (L bits a, b) bool : b <= a;  
h) op = = (L bits a, int b) L bits : if abs a <= b then 
   L bits a := a; to abs b do (b > 0 \land for i from 0 to L bits width do 
   (L F of a) [i] := (L F of b) [i]); L bits width := 
   false); for i from L bits width - 1 to 0 do (L F of a) [i] := false);  
   L int a := a;  
i) op abs = (L bits a) L int : (L int a := L 0; for i to L bits width do 
   c = L 0 \land for i from 0 to L bits width do 
   (L F of a) [i] := (L 0) [i]; (L F of a) [i] := false);  
   L int a := a;  
j) op bin = (L int a) L bits : if a > 0 then L int b := a; L bits a;  
   (L F of a) [i] := odd b; b := b \lor L 2); c \Fi;  
k) op = = (L int a, L bits b) bool : (L F of b) [a];
10.2.8. continued

1) \( \text{op} \ L \ 	ext{bitb} = (l : 1) \) \( \text{bool} \ a) \) \( L \ 	ext{bits} : (\text{int} n = \lfloor a; (n \leq L \ 	ext{bits width}) \ \\
\text{bit} a; \text{for} i \text{ to } L \ 	ext{bits width} \text{ do } (L \ F \ of \ a)[i] := \\\n(t \leq L \ 	ext{bits width} - n \ | \ false \ | a[i - L \ 	ext{bits width} + n]); a)) \);

m) \( \text{op} \neg = (L \ 	ext{bits} a) \) \( L \ 	ext{bits} : (L \ 	ext{bits} a; \text{for} i \text{ to } L \ 	ext{bits width} \text{ do } \\\n(L \ F \ of \ a)[i] := \neg (L \ F \ of \ a)[i]; a) \);

10.2.9. Bytes and associated operations

a) \( \text{struct} \ L \ 	ext{bytes} = ([1 : L \ 	ext{bytes width}] \ \\
\text{char} \ L \ F) ; \) {See 10.2.8.a and \ref{10.1.1}}

b) \( \text{op} \ R = (L \ 	ext{bytes} a, b) \) \( \text{bool} : (\text{string} : a) \ R (\text{string} : b) ; \)

c) \( \text{op} \ 0 = (L \ 	ext{bytes} a, b) \) \( \text{char} : (L \ F \ of \ b)[a] ; \)

d) \( \text{op} \ \text{lctb} = (\text{string} a) \) \( L \ 	ext{bytes} : (\text{int} n = \lfloor a; (n \leq L \ 	ext{bytes width}) \ \\
\text{bytes} a; \text{for} i \text{ to } L \ 	ext{bytes width} \text{ do } \\\n(L \ F \ of \ a)[i] := (i \leq n \ | \ a[i] \ | \ \text{null character}); a)) ; \)

10.2.10. Strings and associated operations

a) \( \text{mode} \ 	ext{string} = [1 : 0 \ \text{flex}] \) \( \text{char} ; \)

b) \( \text{op} \ < = (\text{string} a, b) \) \( \text{bool} : (\text{int} m = \lfloor a; n = \lfloor b; \ \\
\text{int} i := 1; \text{bool} a; \ \\
(t < i \ | \ n \geq 1 \ | \ a; (c := a[i] = b[i] ?) ; (i := i + 1) \leq p \ | \ e); \ \\
(a \ | \ m \leq n \ | \ a[i] \leq b[i]) \)) \);

c) \( \text{op} \ \leq = (\text{string} a, b) \) \( \text{bool} : \neg (b < a) ; \)

d) \( \text{op} \ = = (\text{string} a, b) \) \( \text{bool} : a = b \ \\
\text{and} b = a ; \)

e) \( \text{op} \ * = (\text{string} a, b) \) \( \text{bool} : \neg (a \neq b) ; \)

f) \( \text{op} \ >= = (\text{string} a, b) \) \( \text{bool} : a \geq b \ \\
\text{and} b \geq a ; \)

g) \( \text{op} \ > = (\text{string} a, b) \) \( \text{bool} : b < a ; \)

h) \( \text{op} \ R = (\text{string} a, \) \( \text{char} \) \( b) \) \( \text{bool} : (\text{string} : a) \ R (\text{string} : b) ; \)

i) \( \text{op} \ R = (\text{char} a, \) \( \text{string} \) \( b) \) \( \text{bool} : (\text{string} : a) \ R (\text{string} : b) ; \)

j) \( \text{op} \ + = (\text{string} a, b) \) \( \text{string} : (\text{int} m = \lfloor a; n = \lfloor b; [1 : m + n] \text{char} ; \ \\
\text{int} m := \lfloor a; \text{int} m + 1 : m + n := \lfloor b; a ;) ; \)

k) \( \text{op} \ + = (\text{string} a, \) \( \text{char} \) \( b) \) \( \text{string} : a + (\text{string} : b) ; \)

l) \( \text{op} \ + = (\text{char} a, \) \( \text{string} \) \( b) \) \( \text{string} : (\text{string} : a) + b ; \)

(The operations defined in b, h and i imply that if \( \text{abs} \ "a" < \text{abs} \ "b" \), then \( "\text{lt}" < "a" ; "\text{lt}" < "b" ; "\text{lt}a" < "\text{lt}b" ; "\text{lt}a" < "\text{lt}b" ; "\text{lt}b" < "\text{lt}b". \)

10.2.11. Operations combined with assignations

a) \( \text{op} \ := = (\text{ref} L \ \text{int} a, L \ \text{int} b) \) \( \text{ref} L \ \text{int} : a := a - b ; \)

b) \( \text{op} \ := = (\text{ref} L \ \text{real} a, L \ \text{real} b) \) \( \text{ref} L \ \text{real} : a := a - b ; \)

c) \( \text{op} \ := = (\text{ref} L \ \text{compl} a, L \ \text{compl} b) \) \( \text{ref} L \ \text{compl} : a := a - b ; \)

d) \( \text{op} \ := = (\text{ref} L \ \text{int} a, L \ \text{int} b) \) \( \text{ref} L \ \text{int} : a := a + b ; \)

e) \( \text{op} \ := = (\text{ref} L \ \text{real} a, L \ \text{real} b) \) \( \text{ref} L \ \text{real} : a := a + b ; \)

f) \( \text{op} \ := = (\text{ref} L \ \text{compl} a, L \ \text{compl} b) \) \( \text{ref} L \ \text{compl} : a := a + b ; \)

g) \( \text{op} \ := = (\text{ref} L \ \text{int} a, L \ \text{int} b) \) \( \text{ref} L \ \text{int} : a := a \times b ; \)

h) \( \text{op} \ := = (\text{ref} L \ \text{real} a, L \ \text{real} b) \) \( \text{ref} L \ \text{real} : a := a \times b ; \)

i) \( \text{op} \ := = (\text{ref} L \ \text{compl} a, L \ \text{compl} b) \) \( \text{ref} L \ \text{compl} : a := a \times b ; \)

j) \( \text{op} \ := = (\text{ref} L \ \text{int} a, L \ \text{int} b) \) \( \text{ref} L \ \text{int} : a := a \div b ; \)

k) \( \text{op} \ := = (\text{ref} L \ \text{real} a, L \ \text{real} b) \) \( \text{ref} L \ \text{real} : a := a \div b ; \)

l) \( \text{op} \ := = (\text{ref} L \ \text{compl} a, L \ \text{compl} b) \) \( \text{ref} L \ \text{compl} : a := a \div b ; \)

m) \( \text{op} \ := = (\text{ref} L \ \text{int} a, L \ \text{int} b) \) \( \text{ref} L \ \text{int} : a := a / b ; \)

n) \( \text{op} \ := = (\text{ref} L \ \text{real} a, L \ \text{real} b) \) \( \text{ref} L \ \text{real} : a := a / b ; \)
10.2.11. continued

\[
\begin{align*}
\text{n)} & \quad \text{op } \varnothing = (\text{ref } L \text{ real } a, L \text{ int } b) \text{ ref } L \text{ real } : a \varnothing (L \text{ real } : b) ; \\
\text{o)} & \quad \text{op } \varnothing = (\text{ref } L \text{ complex } a, L \text{ int } b) \text{ ref } L \text{ complex } : a \varnothing (L \text{ complex } : b) ; \\
\text{p)} & \quad \text{op } \varnothing = (\text{ref } L \text{ complex } a, L \text{ real } b) \text{ ref } L \text{ complex } : a \varnothing (L \text{ complex } : b) ; \\
\text{q)} & \quad \text{op } +:= = (\text{ref } \text{ string } a, \text{ string } b) \text{ ref } \text{ string } : a += a + b ; \\
\text{r)} & \quad \text{op } +:= = (\text{string } a, \text{ ref } \text{ string } b) \text{ ref } \text{ string } : b += a + b ; \\
\text{s)} & \quad \text{op } +:= = (\text{char } a, \text{ ref } \text{ string } b) \text{ ref } \text{ string } : b += a + b ; \\
\text{t)} & \quad \text{op } +:= = (\text{char } a, \text{ ref } \text{ string } b) \text{ ref } \text{ string } : b := a + b ;
\end{align*}
\]

10.3. Standard mathematical constants and functions

\[
\begin{align*}
\text{a)} & \quad \text{L real } L \pi = \varnothing \text{ a } L \text{ real value close to } \pi ; \text{ see Math. of Comp. v. 16, } 1965, \text{ pp. 80-99 } \varnothing ; \\
\text{b)} & \quad \text{proc } L \text{ sqrt } = (L \text{ real } x) L \text{ real } : \varnothing \text{ if } x \approx L 0, \text{ a } L \text{ real value close to } \text{ the square root of } 'x' \varnothing ; \\
\text{c)} & \quad \text{proc } L \text{ exp } = (L \text{ real } x) L \text{ real } : \varnothing \text{ a } L \text{ real value, if one exists, close to the exponential function of } 'x' \varnothing ; \\
\text{d)} & \quad \text{proc } L \text{ ln } = (L \text{ real } x) L \text{ real } : \varnothing \text{ a } L \text{ real value, if one exists, close to the natural logarithm of } 'x' \varnothing ; \\
\text{e)} & \quad \text{proc } L \text{ cos } = (L \text{ real } x) L \text{ real } : \varnothing \text{ a } L \text{ real value close to the cosine of } 'x' \varnothing ; \\
\text{f)} & \quad \text{proc } L \text{ arccos } = (L \text{ real } x) L \text{ real } : \varnothing \text{ if abs } x \leq L 1, \text{ a } L \text{ real value close to the inverse cosine of } 'x', \text{ abs } L 0 \leq L \text{ arccos } (x) \leq L \pi \varnothing ; \\
\text{g)} & \quad \text{proc } L \text{ sin } = (L \text{ real } x) L \text{ real } : \varnothing \text{ a } L \text{ real value close to the sine of } 'x' \varnothing ; \\
\text{h)} & \quad \text{proc } L \text{ arcsin } = (L \text{ real } x) L \text{ real } : \varnothing \text{ if abs } x \leq L 1, \text{ a } L \text{ real value close to the inverse sine of } 'x', \text{ abs } L 0 \leq L \text{ arcsin } (x) \leq L \pi / L 2 \varnothing ; \\
\text{i)} & \quad \text{proc } L \text{ tan } = (L \text{ real } x) L \text{ real } : \varnothing \text{ a } L \text{ real value, if one exists, close to the tangent of } 'x' \varnothing ; \\
\text{j)} & \quad \text{proc } L \text{ arctan } = (L \text{ real } x) L \text{ real } : \varnothing \text{ a } L \text{ real value close to the inverse tangent of } 'x', \text{ abs } L 0 \leq L \text{ arctan } (x) \leq L \pi / L 2 \varnothing ; \\
\text{k)} & \quad \text{proc } L \text{ real } L \text{ random } = L \text{ last random } : \varnothing \text{ the next pseudo-random } L \text{ real } \text{ value after } L \text{ last random from a uniformly distributed sequence on the interval } [L 0, L 1] \varnothing ; \\
\text{l)} & \quad L \text{ real } L \text{ last random } := L .0 ;
\end{align*}
\]

10.4. Synchronization operations

\[
\begin{align*}
\text{a)} & \quad \text{struct } \text{ sema } = (\text{ref } \text{ int } F) ; \\
\text{b)} & \quad \text{op } /= (\text{int } a) \text{ sema } : (\text{sema } a; F \text{ of } a := \text{int } := a; a) ; \\
\text{c)} & \quad \text{op } ++ = (\text{sema edager}) : (\text{ref } \text{ int } \text{ dijkstra } = F \text{ of edager}; \text{do } (\text{if } \text{ dijkstra } \geq 1 \text{ then } \text{ dijkstra } := 1; \text{ else } \text{ skip}) \text{ if the closed-statement replacing this comment is contained in a unitary-phrase which is a constituent unitary-phrase of the smallest collateral-phrase, if any, beginning with a parallel-symbol and containing this closed-statement, then the elaboration of that unitary-phrase is halted (6.0.2.c); otherwise, the further elaboration is undefined a \text{ ft}) ; L \text{ skipt}) ; \\
\text{d)} & \quad \text{op } += = (\text{sema edager}) : (\text{ref } \text{ int } \text{ dijkstra } = F \text{ of edager}; \text{dijkstra } += 1; \text{ a the elaboration is resumed of all phrases whose elaboration is halted because the name possessed by dijkstra referred to a value smaller than one a}) ;
\end{align*}
\]

(See 2.2.5; for the use of + and ++, see E.W. Dijkstra, Cooperating Sequential Processes, contained in Programming Languages, Genuys, F. (ed.), London, etc., Academic Press, 1968; see also 11.13.)
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) union % simplout = (1L int, 1L real, 1L comp1, bool, char, string);
b) union % outtype = (10 L int, 10 L real, 10 L bool, 10 L char, 10 L outstruct);
c) mode % outstruct = a on actual-declarer specifying a mode united from (4.4.3.a) all modes, except those specified by tamref and by ansa, which are structured from (2.2.4.1.d) only modes from which the mode specified by outtype is united a;
d) struct % tamref = (string F1); (See the remarks under 5.5.5.)
e) union % intype = (ref D L int, ref D L real, ref D L bool, ref D L char, ref D L outstruct);

10.5.0.2. Straightening

a) op % straightout = (outtype x) [] simplout:
   \( a \) the result of "straightening" \( \rightarrow a \);
b) op % straightin = (intype x) [] ref simplout:
   \( a \) the result of straightening \( \rightarrow a \);

c) The result of straightening a given value \( V \) is a multiple value obtained in the following steps:
Step 1: If \( V \) is (refers to) a value from whose mode that specified by simplout is united, then the result is a new instance of a multiple value composed of a descriptor consisting of an offset \( I \) and one quintuple \( (1,1,1,1,1) \) and \( V \) as its only element, and Step 1 is taken;
Step 2: If \( V \) is (refers to) a multiple value, then letting \( n \) stand for the number of elements of that value, and \( y_{i} \) for the result of straightening its \( i \)-th element, Step 3 is taken; otherwise, letting \( n \) stand for the number of fields of \( (\text{of the value referred to by} \ V) \), and \( y_{j} \) for the result of straightening its \( j \)-th field, Step 3 is taken;
Step 3: If \( V \) is not (isa) a name, then, letting \( m_{i} \) stand for the number of elements of \( y_{i} \), the result is a new instance of a multiple value composed of a descriptor consisting of an offset \( I \) and one quintuple \( (1, m_{1} + \ldots + m_{k}, 1, 1, 1) \) and elements, the \( k \)-th of which, where \( k = m_{1} \), \( \ldots \), \( n \) and \( j = 1, \ldots, m_{k} \); Step 4: If \( V \) is not (isa) a name, then the mode of the result is 'row of' ('row of reference to') followed by the mode specified by simplout.
10.5.1. Channels and files

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

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

c) All information on a given channel is to be found in a number of backfiles. A backfile (10.5.1.1.b) comprises a three-dimensional array of integers (bytes of information), the book of the backfile; the lower bounds of the book are all 1, the upper bounds are nonnegative integers, maxpage, maxline and maxchar of the backfile; furthermore, the backfile comprises the position of the "end of file", i.e., the page number, line number and char number up to which the backfile is filled with information, the current position and the "identification string" of the backfile.

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

Examples:

a) In a certain implementation, channel 6 is a line printer. It has no input information, chainbackfile [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 both the current position and the end of file at [1, 1, 1] and next equal to nil. All elements of the book are left undefined.

b) Channel 4 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 chainbackfile [4] is initialized to refer to the first backfile of a chain of 32 backfiles, the last one having next equal to nil. Each of those backfiles has an end of file at position [2, 1, 1].

c) Channel 20 is a tape unit. It can accommodate one tape at a time; one input tape is mounted and another tape laid in readiness. Here, chainbackfile [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.

e) A file (10.5.1.2.a) is a structured value which comprises a reference to a backfile, and the information necessary for the transport routines to work with that backfile. A backfile is associated with a file by means of open (10.5.1.2.b), create (10.5.1.2.c) or establish (10.5.1.2.d). A given channel can accommodate a certain number (10.5.1.1.o) of backfiles at any stage of the elaboration. The association is ended by means of close (10.5.1.2.e), lock (10.5.1.2.f) or scratch (10.5.1.2.u).
<table>
<thead>
<tr>
<th>properties</th>
<th>card reader</th>
<th>card punch</th>
<th>magnetic tape unit</th>
<th>line printer</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset possible</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>set possible</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>get possible</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>put possible</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>bin possible</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>idf possible</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>reidf possible</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>max page</td>
<td>1</td>
<td>1</td>
<td>very large</td>
<td>very large</td>
</tr>
<tr>
<td>max line</td>
<td>large</td>
<td>very large</td>
<td>16</td>
<td>large</td>
</tr>
<tr>
<td>max char</td>
<td>72</td>
<td>80</td>
<td>84</td>
<td>large</td>
</tr>
<tr>
<td>stand comp</td>
<td>a 48- or 64-character code</td>
<td>64-char code</td>
<td>some code</td>
<td>line-pr code</td>
</tr>
<tr>
<td>max nmb files</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>properties</th>
<th>magnetic disc</th>
<th>magnetic drum</th>
<th>paper tape reader</th>
<th>tape punch</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset possible</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>set possible</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>get possible</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>put possible</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>bin possible</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>idf possible</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>reidf possible</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>max page</td>
<td>200</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>max line</td>
<td>16</td>
<td>1</td>
<td>very large</td>
<td>very large</td>
</tr>
<tr>
<td>max char</td>
<td>128</td>
<td>524288</td>
<td>256</td>
<td>80</td>
</tr>
<tr>
<td>stand comp</td>
<td>some code</td>
<td>some code</td>
<td>some code</td>
<td>5-hole code</td>
</tr>
<tr>
<td>max nmb files</td>
<td>10</td>
<td>4</td>
<td>32</td>
<td>lathe code</td>
</tr>
</tbody>
</table>

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

gg) When a file is "established" on a channel, then a backfile is generated (8.5) with a book of the given size, the given identification string and both the current position and the end of file at [1, l, l]; when a file is "created" on a channel, then a file is established with a backfile the book of which has the maximum size for the channel and an empty string as its identification string.

hh) When a file is "scratched", then its associated backfile is obliterated.

ii) When a file is "closed" ("locked"), then its associated backfile is attached to the chain referenced by chainbfile (lookedbfile) of the channel. Another file may (No file can) now be opened with this backfile by a call of open.

jj) The identification string of the backfile of a file opened on a channel for which the value of reidf possible is true may be changed by a call of reidf.

kk) A file comprises some fields of the mode 'procedure boolean', 'procedure with reference to character parameter boolean' or 'procedure with integral parameter boolean', routines which are called when in transput certain error situations arise. After opening or creating a file, the routines provided yield the value false when called, but the programmer may assign other routines to those fields. If the elaboration of such a routine is terminated, then the transput routine which called it can take no further action; otherwise, if it yields the value true, then it is assumed that the error situation has been remedied in some way and, if possible, transput goes on, but if it yields the value false, then undefined is called, i.e., some sensible system action is taken (rr). These routines are:

a) logical file end , which is called when during input from a file on a sequential channel the end of file of its backfile is passed. If the routine yields the value true, then transput goes on, and if it yields false, then some sensible action is taken. Example: The programmer wishes to count the number of integers on his input tape. The file intape was opened in a surrounding range. If he writes

```
begin int n := 0; logical file end of intape := go to f;
do (get (intape, loc int); n := l); f: print (n)
end .
```

then the assignment to the field of intape violates the scope restrictions (; the scope of the routine (: go to f)) is smaller than the scope of intape), so he has to write

```
begin int n := 0; file auxin := intape;
logical file end of auxin := go to f;
do (get (auxin, loc int); n := l); f: print (n)
end .
```
10.5.1. continued 2

b) physical file end, which is called when the max page, the max line or the max char of the backfile of a file is exceeded. If the routine yields the value true, then transput goes on, and if it yields false, then some sensible action is taken.

Example:
The programmer wishes automatically to give a new line at the end of a line and a new page at the end of a page on his file f:

```plaintext
proc bool new line page = bool : ((line ended (f) | new line (f));
(page ended (f) | new page (f)); true);
```

c) char error, which is called when, during formatted input, a character is read which does not agree with the frame specifying it (5.5.1.1.m) or when, during input, at the current position an uninterpretable character is present (10.5.1.11), with a reference to a character, suggested as a replacement. The routine provided by the programmer may give some other character than the suggested one. If the routine yields true, then that suggested character as possibly modified by the routine is used, and, if it yields false, then some sensible action is taken.

Example:
The programmer wishes to print a list of all such disagreements. He assigns to the field char error of his file f

```plaintext
((ref char sugg) bool : (char k; backspace (f);
int p = page number (f), l = line number (f), c = char number (f);
get (f, k); prinf ((new line, "at", p, l, c, "present", "", k, "","",_suggested, "", sugg, "", "); true));
```

d) value error, which is called when, during formatted transput, an attempt is made to transput a value under control of a picture with which it is not compatible, or when the number of frames is not sufficient. If the routine yields true, then the current value and picture are skipped, i.e., transput goes on at 5.5.1.4.d.Step 5; if the routine yields false, then first, on output, the value is output by put, and next some sensible action is taken.

e) format end, which is called when, during formatted transput, the format is exhausted while still some value remains to be transput. If the routine yields true, then transput goes on (so the routine must have provided a new format for the file), and, if the routine yields false, then the current format is repeated, i.e., the first picture again is made to be the current picture of the file.

f) other error, which is called with some actual-integral-parameter, when during transput some other error situation arises. No call of this routine occurs explicitly in the standard-prelude, and neither the meaning of its actual-parameter nor that of the value yielded, is defined in this Report. This routine may, in some implementation, be called when an incorrigible hardware error occurs which makes transput involving this file impossible. (The programmer may provide a routine which then closes the file and opens it on some other channel.)

11) The conv of a file is used by the transput routines in the conversion of characters to and from integers in the book of the bfile of the file. After opening, creating or establishing a file, stand conv of the channel is used, but some other "conversion key" may be provided by the programmer by a call of make conv (10.5.1.2.e).
10.5.1. continued

On output, the character is converted to that integer, if any, in the conversion key, whose ordinal number is the integral equivalent of that character; what action is taken when an attempt is made to convert a character with an integral equivalent exceeding the upper bound of the conversion key, is left undefined; on input, the given integer is converted to that character, if any, whose integral equivalent is the lowest ordinal number for which the element of the conversion key is equal to that given integer; if no such character exists, then char error is called with a space (parity error, nonexistent code).

The term of a file is used in reading strings of a variable number of characters, where any of the characters of term serves as a terminator (see 5.5.1.jj. and 10.5.2.2.dd). This terminator string may be provided by the programmer. Furthermore, when reading outside the file, physical file end of the file is called, and if that does not cause the position to be within the file, that also serves as a terminator.

On a channel for which the value of reset possible is true, a file may be "reset", causing its position to be [1, 1, 1]. On a sequential access file, the end of file remains at the position up to which the backfile contains information, but when after resetting any output is done, then the end of file is first set at the current position.

On a random access channel, a file may be "set", causing its position to be the given position.

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

On files opened on a sequential access channel for which put possible and get possible both possess the value true, nonbinary input and output may be alternated, but it is not allowed to read past the end of file.

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

Example:

```pascal
begin file 1, 2; [1: 10000] int x, int n; open (f1, channel 1); f2 := f1; \ now f1 and f2 can be used interchangeably \ make conv (f1, flexocode); make conv (f2, telexode); \ now f1 and f2 use different codes; flexocode and telexode are defined in the library-declaration for this implementation \ reset (f1); \ consequently, f2 is reset too \ for i while ~logical file ended (f1) do (n := i; get (f1, x[i])); \ too bad if there are more than 10000 integers in the input \ reset (f1); for i to n do put (f2, x[i]); reset (f2); close (f2) \ if 1 is now closed too \ end }
```
10.5.1.1. Channels

a) \texttt{int \ nmb\ channels\ =\ 0; an integral-clause indicating the number of trans-}
\texttt{put channels in the implementation \&;}
b) \texttt{struct \ \%\ bfile = ([1 : 0 flex, 1 : 0 flex, 1 : 0 flex] \ int \ book,}
\texttt{int \ ipage, iline, iline, page, line, char, max page, max line,}
\texttt{max char, string idf, ref \ bfile \ next);}
c) \texttt{[1 : nmb\ channels] \ ref \ bfile \ \%\ chainbfile := \ a \ some \ appropriate \ initial-
\texttt{ization \ (see 10.5.1.dd) \ &;}
d) \texttt{[1 : nmb\ channels] \ bool \ reset\ possible = \ a \ row-of-boolean-clause, in-
\texttt{dicating which of the physical devices corresponding to the channels}
\texttt{allow resetting \ (e.g., rewinding of a magnetic tape) \ \&;}
e) \texttt{[1 : nmb\ channels] \ bool \ set\ possible = \ a \ row-of-boolean-clause, indi-
\texttt{cating which devices can be accessed at random \ &;}
f) \texttt{[1 : nmb\ channels] \ bool \ get\ possible = \ a \ row-of-boolean-clause, indi-
\texttt{cating which devices can be used for input \ &;}
g) \texttt{[1 : nmb\ channels] \ bool \ put\ possible = \ a \ row-of-boolean-clause, indi-
\texttt{cating which devices can be used for output \ &;}
h) \texttt{[1 : nmb\ channels] \ bool \ bin\ possible = \ a \ row-of-boolean-clause; in-
\texttt{dicating which devices can be used for binary transport \ \&;}
i) \texttt{[1 : nmb\ channels] \ bool \ idf\ possible = \ a \ row-of-boolean-clause, indi-
\texttt{cating which devices backfiles have an identification \ \&;}
j) \texttt{[1 : nmb\ channels] \ bool \ reidf\ possible = \ a \ row-of-boolean-clause, in-
\texttt{dicating which devices backfiles allow reidentification \ \&;}
k) \texttt{[1 : nmb\ channels] \ int \ max\ page = \ a \ row-of-integral-clause, giving the}
\texttt{maximum number of pages per file for the channel \ \&;}
l) \texttt{[1 : nmb\ channels] \ int \ max\ line = \ a \ row-of-integral-clause, giving the}
\texttt{maximum number of lines per page \ \&;}
m) \texttt{[1 : nmb\ channels] \ int \ max\ char = \ a \ row-of-integral-clause, giving the}
\texttt{maximum number of characters per line \ \&;}

1) \texttt{[1 : nmb\ channels] \ struct \ (proc int \ f) \ stand\ conv = \ a \ clause giving}
\texttt{the standard conversion keys for the channels \ (other conversion keys}
\texttt{may be provided by the library-prelude) \ \&;}

1) \texttt{[1 : nmb\ channels] \ int \ max\ nmb\ files = \ a \ row-of-integral-clause, giv-
\texttt{ing the maximum number of files the channels can accommodate \ \&;}

\texttt{[1 : nmb\ channels] \ int \ \%\ nmb\ opened\ files;}
\texttt{for \ i \ to \ nmb\ channels \ do \ nmb\ opened\ files \ [i] := 0;}
\texttt{[1 : nmb\ channels] \ ref \ bfile \ \%\ lockedbfile;}
\texttt{for \ i \ to \ nmb\ channels \ do \ lockedbfile \ [i] := nil;}
\texttt{\ proc\ file\ available = (int\ channel) \ bool :}
\texttt{\ nmb\ opened\ files \ [channel] < max\ nmb\ files \ [channel];}

10.5.1.2. Files

\texttt{[1 : nmb\ channels] \ bfile = (ref \ bfile \ \%\ bfile, int \ chan, ref \ int \ \%\ forp,}
\texttt{ref \ bool \ \%\ state\ def, \% state\ get, \% state\ bin, \% opened,}
\texttt{ref \ string \ \%\ format, \% string\ term, [0 : 0 flex] \ int \ \%\ conv,}
\texttt{proc \ bool \ logical\ file\ end, \ physical\ file\ end, \ format\ end,}
\texttt{value\ error, \ proc \ (ref \ chan) \ bool \ char\ error,}
\texttt{proc \ (int) \ bool \ other\ error);}
b) proc open = (ref file file, string idf, int oh) : 
    if file available (oh) 
    then ref ref bf file := chaintbf [oh]; 
    while (ref bf file := bf) != nil do 
        (idf of bf := idf v idf = "m" v - idf possible [oh]) 
        l | bf := next of bf; undefined. 
    l: file := (bf, oh, int := 0, bool := false, bool, bool, bool := true, 
        nil, "m", F of standom [oh], false, false, false, false, 
        (ref char a) bool := false) (skip); 
    (ref ref bf file := bf) := next of bf; mb opened files [oh] += 1 
    else undefined ft; 

c) proc create = (ref file file, int oh) : 
    establish (file, max page [oh], max line [oh], max char [oh], oh) ; 

d) proc establish = (ref file file, string idf, int mp, ml, mo, oh) : 
    if file available (oh) v mp <= max page [oh] v ml <= max line [oh] v 
        mo <= max char [oh] 
    then bf file := (l1 := mp, l := ml, m := mo) int, i, i, i, i, i, i, i, 
        ml, mo, idf, nil) ; 
    file := (bf file := bf, oh, int := 0, bool := false, bool, 
        bool, bool := true, nil, "m", F of standom [oh], false, 
        false, false, false, (ref char a) bool := false) (skip); 
    mb opened files [oh] += 1 
    else undefined ft; 

e) proc set = (file file, int p, l, a) : 
    if set possible [chan of file] ^ opened of file 
    then page of bf file of file := p; line of bf file of file := l; 
        char of bf file of file := a; check pio (file) 
    else undefined ft; 

f) proc reset = (file file) : 
    if reset possible [chan of file] ^ opened of file 
    then page of bf file of file := 1; line of bf file of file := 1; 
        char of bf file of file := 1; state def of file := false 
    else undefined ft; 

g) proc pio check = (file file) : if opened of file 
    then ! (logical file ended (file) | logical file end of file | 
        line ended (file) v page ended (file) v file ended (file) | 
        physical file end of file | true | undefined) 
    else undefined ft; 

h) proc line ended = (file file) bool : (opened of file | 
    int a = char of bf file of file; 
    a <= 0 v a > max char of bf file of file) ; 

i) proc page ended = (file file) bool : (opened of file | 
    int l = line of bf file of file; 
    l <= 0 v l > max line of bf file of file) ; 

j) proc file ended = (file file) bool : (opened of file | 
    int p = page of bf file of file; 
    p <= 0 v p > max page of bf file of file) ; 

k) proc logical file ended = (file file) bool : (opened of file | 
    if set possible [chan of file] ^ state def of file ^ state get of file | 
    bf file b = bf file of file; int p = page of b, tp = page of b, 
    l = line of b, ll = line of b, a = char of b, lc = tc of char of b; 
    (p < tp | false | p > tp | true | i < ll | false | i > ll | true | 
    a zuc) | false) ;
10.5.1.2. continued 2

1) procs % get string = (file file, ref [1:either] char s) : 
   if get possible [chan of file] & opened of file 
      then ref int p = page of bf of file, l = line of bf of file, 
          c = char of bf of file; 
       if -set possible [chan of file] then state def of file 
       then (state bin of file = undefined) fi; 
       state def of file := state get of file := true; 
       state bin of file := false; 
       for i to | s do (check plo (file)); 
       for j from 0 to | conv of file do 
          (*conv of file) {j} = (book of bf of file) [p, l, c] | 
          s [j] := repf j; e); 
       c := "\n"; 
       s [l] := ((char error of file) (k) | k | undefined; ";
       e: c +== 1) 
   else undefined fi;

m) procs % put string = (file file, string s) : 
   if put possible [chan of file] & opened of file 
      then ref int p = page of bf of file, l = line of bf of file, 
          c = char of bf of file; 
       if -set possible [chan of file] then state def of file 
       then (state bin of file = undefined) fi; 
       state get of file := state bin of file := false; 
       state def of file := true; 
       for i to | s do (check plo (file)); 
       (book of bf of file) [p, l, c] := (conv of file) [abs s [i]];
       c +== 1; 
       (p = page of bf of file & l = line of bf of file | 
         (c = char of bf of file) (k) | k | undefined; ");
       (p = page of bf of file := p ; line of bf of file := l; 
         char of bf of file := c)) 
   else undefined fi;

n) procs char in string = (char c, ref int i, string s) bool : 
   (for k to | s do (c = s [k] | i := k; false, i; true) ;

o) procs space = (file file) : 
   (char of bf of file +== 1; check plo (file));

p) procs backspace = (file file) : 
   (char of bf of file -== 1; check plo (file));

q) procs new line = (file file) : (line of bf of file +== 1; 
   char of bf of file := 1; check plo (file));

r) procs new page = (file file) : (page of bf of file +== 1; 
   line of bf of file := char of bf of file := 1; 
   check plo (file));

s) procs close = (file file) : (opened of file | 
   int oh = chan of file; next of bf of file := chan bf file [oh]; 
   chan bf file [oh] := bf of file; opened of file := false; 
   mbb opened files [oh] -== 1) ;

t) procs look = (file file) : (opened of file | 
   int oh = chan of file; ref bf bf bf = bf of file; 
   page of bf := line of bf := char of bf := 1; 
   next of bf := locked bf file [oh]; locked bf file [oh] := bf; 
   opened of file := false; mbb opened files [oh] -== 1) ;

u) procs scratch = (file file) : (opened of file | 
   opened of file := false; mbb opened files [chan of file] -== 1) ;
10.5.1.2. continued 3

v) \texttt{proc char number \{ file f \} int} : \texttt{(opened of f | char of bfile of f)} ;
w) \texttt{proc line number \{ file f \} int} : \texttt{(opened of f | line of bfile of f)} ;
x) \texttt{proc page number \{ file f \} int} : \texttt{(opened of f | page of bfile of f)} ;
y) \texttt{proc \% undefined} : \texttt{c some sensible system action} (10.5.1.xrr \{ f \}) ;
z) \texttt{proc make conv \{ ref file f, struct \{ proc [] int f \} a \} : conv of f := \texttt{f of a} ;

aa) \texttt{proc relid = \{ file f, string idf \} : (relid possible \{ char of f \} | idf of bfile of f := idf)} ;

10.5.1.3. Standard channels and files

a) \texttt{int stand in channel = c an integral-clause such that the value of get possible [stand in channel] is true c} ;
b) \texttt{int stand out channel = c an integral-clause such that the value of put possible [stand out channel] is true c} ;
c) \texttt{int stand back channel = c an integral-clause such that the values of 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 c} ;
da) \texttt{file \% f ; open (f,, stand in channel) ; file stand in = f} ;
e) \texttt{open (f,, stand out channel) ; file stand out = f} ;
f) \texttt{open (f,, stand back channel) ; file stand back = f} ;

(Certain "standard files" (d, e, f) need not (and cannot) be opened by the programmer, but are opened for him in the standard-prelude; print (10.5.2.1.a) can be used for output on stand out, read (10.5.2.2.a) for input from stand in, and write bin (10.5.4.1.a) and read bin (10.5.4.2.a) for transput involving stand back. The programmer need not close these standard files, since they are locked in the standard-postlude.)

10.5.2. Formatless transput

10.5.2.1. Formatless output

(For formatless output, print and put can be used. The elements of the given value of the mode specified by \{ union (output, proc (file)) \} are treated one after the other; if an element in of the mode specified by proc (file) (i.e., a "layout procedure"), then it is called with the file as its parameter; otherwise, it is straightened (10.5.0.2), and the resulting values are output on the given file one after the other, as follows:
a) If the mode of the value is specified by L int, then first, if there is not enough room on the line for L int width + 2 characters, then this room is made by giving a new line and, if the page is full, giving a new page; next, when not at the beginning of a line, a space is given and the value is edited as if under control of the picture \{ n(L int width - 1)\hd \}.  
bb) If the mode of the value is specified by L real, then first, if there is not enough room on the line for L real width + L expwidth + 5 characters, then this room is made; next, when not at the beginning of a line, a space is given, and the value is edited as if under control of the picture \{ n(L real width - 1)den(L expwidth - 1)\hd \}.  

\texttt{d.n}(\texttt{Lreal width - 1})\texttt{den(L expwidth - 1)}\texttt{\%d}.
\[
\begin{align*}
L \text{ int string} & : +\text{DDDDDDDDDDDD} \\
& \hspace{1cm} \downarrow \text{w} \\
L \text{ dec string} & : +\text{DDDDDD.DDDDD} \\
& \hspace{1cm} \downarrow \text{w} \\
L \text{ real string} & : +\text{DDDDDD.DDDDD}10^+\text{DDD} \\
& \hspace{1cm} \downarrow \text{w}
\end{align*}
\]

TABLE II: Display of the values of \(L\) int string, \(L\) dec string and \(L\) real string (10.5.2.1.c,e,d)

frame

[1] type \((1 = \text{integer}, 2 = \text{real fixed}, 3 = \text{real floating}, \)
\(4 = \text{complex fixed}, 5 = \text{complex floating}, 6 = \text{string}, \)
\(7 = \text{integer choice}, 8 = \text{boolean}, 9 = \text{character})\);

[2] radix \((2, 4, 8, 10 \text{ or } 16)\);

[3] sign \((0 = \text{no sign frame}, 1 = \text{sign frame } +, 2 = \text{sign frame } -)\);

[4] number of digits before point; if type = 1, then \(w - 1\), else if
\(w = 2 \text{ or } 4\), then \(w - d - 2\), else if type = 3 \text{ or } 5, then
\(w - d - e - 4\), or, if type = 6, then the number of characters in the
string if this is constant, or 0, if this number is variable;

[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 in exponents; if type = 3 or 5, then \(e\);


TABLE III: Significance of the elements of frame (10.5.3.1)
cc) If the mode of the value is specified by \texttt{L compl}, then first, if there is not enough room on the line for $2 \times (L \text{ real width} + L \exp width) + 11$ characters, then this room is made; next, when not at the beginning of a line, a space is given, and the value is edited as if under control of the picture \texttt{d.n}(L \text{ real width} + 1)\text{den}(L \text{ expwidth} - 1)sad\text{ad}.n(L \text{ real width} - 1)\text{den}(L \text{ expwidth} - 1)sad.

dd) If the mode of the value is specified by \texttt{[ ] char}, then its elements are written one after the other.

e) If the mode of the value is specified by \texttt{char}, then first, if the line is full, then room is made, and next the character is written.

ff) If the mode of the value is specified by \texttt{bool}, then, if the value is \texttt{true} (false), then the character possessed by the flip- (\texttt{flip-}) symbol is output as \texttt{ee}.

a) \texttt{proc print = ([] union (outtype, proc (file)) x) : put (stand out, x) ;}

b) \texttt{proc put = (file file, [i] union (outtype, proc (file)) x) :}
   \texttt{begin outtype op; proc (file) pf;}
   \texttt{for i to x do (ot := x [i]; pf := x [i] op pf (file) | [i : simpout y = straightout ot;}}
   \texttt{for j to y do (string s, bool b, char k;}
   \texttt{(L int i; [i := y [j] |}
   \texttt{s := L int string (i, L int width + 1, 10);}
   \texttt{sign supp zero (s, i, L int width)());}
   \texttt{(L real x; [x := [j] | s := L real conn (x))());}
   \texttt{(L compl x; [x := [j] | s := L compl conn (x))());}
   \texttt{(b := y [j] | put (file, b | "1" | "0")());}
   \texttt{(k := [j] | nextplo (file); put string (file, k); end);}
   \texttt{(s := [j] | put string (file, s); end);}
   \texttt{ref int c = char of file of file; int a1 = c, n = L s;}
   \texttt{a := (a1 = 1 | n | n + 1);}
   \texttt{(line ended (file) | nextplo (file) | c := a1);}
   \texttt{put string (file, (c = 1 | s | _ + s));}
   \texttt{end: skip);}

c) \texttt{proc L int string = (L int x, int w, r) string : (r > 1 \land r < 17 |}
   \texttt{string c := ; L int n := abs x; L int lr = k r;}
   \texttt{for i to w - 1 do (dig char (S (n := ir)); o := c; n := ir);}
   \texttt{(n = L 0 | (x z L 0 | _ + _ + c | _ + _ + _ + n);}

d) \texttt{proc L real string = (L real x, int w, d, e) string :}
   \texttt{(d > 0 \land e > 0 \land d + e + 4 \leq w |}
   \texttt{L real g = L 10 + (w - d - e - 4); L real h = g \times L 1;}
   \texttt{L real y := abs x; int p := 0; while y z g do (y := L 10; p := 1);}
   \texttt{(y = L 0 | while y < h \land p - 1 \leq 10 + e do (y := L 10; p := 1));}
   \texttt{(y + E 0, d = L 1 \land d + e + g | y := h; p := 1);}
   \texttt{L deo string ((x z 0 | y | _ - y); w - e - 3, d) \times _ + _ + int string (p, e + 1, 10));}

e) \texttt{proc L deo string = (L real x, int w, d) string :}
   \texttt{(abs x L 10 + (w - d - 2) \land d z 0 \land d + 2 \leq w | string s := ;}
   \texttt{L real y := (abs x L 0, d + 1) \times L 1 + (w - d - 2);}
   \texttt{to w - 2 do s := dig char (int a = S (y := L 10); y := (a);)
   \texttt{(s z 0 | _ + _ + s | _ + s | _ + _ + _ + _;}(x z 1 | _ - w - d - 2) + _ + s | w - d - 1));}
fa) \texttt{proc% dig char = (int x) char : ("0123456789abcdef" [x + 1]);}

{In connection with 10.5.2.1.c,d,e, see Table II.}
10.5.2.1. continued 2

g) proc % sign supp zero = (ref string a, int l, w) :
   for i from l + 1 to w while a [i] = "\n" do
      a [i] := a [i - 1]; a [i - 1] := "\n"
   od;

h) int L int width = (int a := 1;
   while L 10 + (a - 1) < L .1 x L max int do a := 1; a od;

i) int L real width = 1 + L (L ln (L small real) / L ln (L 10));

j) int L exp width =
   L + L (L ln (L ln (L max real)) / L ln (L 10)) / L ln (L 10);

k) proc % L real conv = (L real a) string :
   string a := L real string (a, L real width + L exp width + 4,
   L real width - 1, L exp width); sign supp zero (a, L real width + 4,
   L real width + L exp width + 3); a od;

l) proc % next pio = (file file) : (opened of file |
   (line ended (file) | new line (file));
   (page ended (file) | new page (file)))

10.5.2.2. Formatless input

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

   a) If the name refers to a value whose mode is specified by L int, then
first, the file is searched for the first character that is not a space
(giving new lines and pages as necessary); next, the largest string is
read from the file that could be indented under control of some picture of
the form n(k3)ad or m(n(1))"n(k3)ad; this string is converted to an inte-
ger by L string int.

   b) If the name refers to a value whose mode is specified by L real, then
first, the file is searched for the first character that is not a space;
next, the largest string is read from the file that could be indented under
control of a picture of the form m(n(1))"n(k3)ad or n(k3)d followed by
n(k3)ad or d, possibly followed by en(k3)"en(k3)"n(k3)ad or
en(k3)"en(k3)"n(k3)ad; this string is converted to a real number by L string
real.

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

   d) If the name refers to a value whose mode is specified by [] char, then,
if both upper and lower state of the value are 1, then as many characters
are read as the value has elements; if not both states are 1, then charac-
ters are read from the line under control of the terminator string refer-
ced by the file (5.5.1,jj, 10.5.1:mm); the string with those characters as its elements is then the resulting value.

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

   f) If the name refers to a value whose mode is specified by bool, then
first, the file is searched for the first character that is not a space;
then a character is read; if this character is that possessed by the flip-
(flop) symbol, then the resulting value is true (false); if the character
is neither of those, then char error of the file is called with (false.)
10.5.2.2. continued

a) proc read = ([] union (intype, proc (file))) =: get (stand in, x) ;
b) proc get = (file file, [1 : [] union (intype, proc (file))]) :
   begin intype rt; proc (file) pf; char k; priority i = 8;
   for i to x do
      if (it := x [i-1]; pf := x [i-1]; pf (file) | [1] ref simput y = straight in it;
      op i = (string a) bool : (outside (file) | false) :
         get string (file, k); char in string (k, loc int, a) |
         true | backspace (file); false);
      op i = (char a) bool : (? (string a)) ;
      op i = (string a, char a) char : (get string (file, k);
         char in string (k, loc int, a) | k | char sug = a;
         ((char error of file) (sugg) | sug | undefined; 0));
      op i = (char s, o) char : (string s ! o;
      proc skip spaces = : while (next plc (file); ? "\n") do skip;
      proc string read dig = string : (string t : "0123456789") 7 \n t;
      while ? "0123456789" do t := k; t);
      proc string read real = string : (string t := read num;
         (? \n t := \n t read dig);
         (? "e\n" t + := \n t + read num); t);
      for j to y do (ref bool bb; ref char ac; ref string ss;)
         (ref int ti; ti := y [j]) |
         (ref int : ti) := L string int (read num, 10))) | |
         (ref int : real xw; (xw := y [j]) |
         (ref L real : xw) := L string real (read real)) |
         (ref L coml as; (as := y [j]) get (file, re of as);|
         skip spaces; \n t := \n t get (file, im of as)); |
         (bb := y [j]) | skip spaces;
         (ref bool : bb) := (? "1" | true | "0" ! "0" = "1");
      (ac := y [j]) | next plc (file); get string (file, 0)); |
      (as := y [j]) : \ as \ as \ get string (file, as [0 1]) |
      string ti; while (outside (file) |
      physical file end of file) ; outside (file) ; false |
      : ? term of file | backspace (file) ; false | true) |
      do t := k; (ref string : as) :=
         t (0 \ as \ as := \ as u 1 | t + \ as | 1) | |
   end ;

c) proc L string int = (string x, int r) L int : |
   (r > 1 & r < 17) | L int tr = K r; int w = [ x ;
   for i from 3 to w do n := n + tr + K (int d = char dig (x [i-1])); |
   (d < r | d); (a [i] = "$" | n := x [i]) := "$" | n = |
   ) ;

da) proc L string real = (string x, L real) : |
   (int i; (char in string ("\n", x, z)) | true |
   char in string ("^e", e, x) | L string dec (x [1 + e - 1]) |
   L 10.0 + string int (x [e + 1 :], 10) | L string dec (x [e]));

e) proc L string dec = (string x, L real) : |
   (int w = [ x; L real r = L 0; |
   int p; (char in string ("^e", p, z)) |
   (1 : w - 2) char s = x [2 : p - 1] + x [p + 1 :];
   for i to w - 2 do r := L 10.0 + p + K (int d = char dig (s [i-1])); |
   (d < r | d); (a [i] = "$" | r := x [i] = "$" | r = |
   L 1.0 + (w - p) | L string dec (a + "$")));
f) proc % char dig = (char x) int := (int i;
   (char in string (x, t, "0123456789abcdef") | i - 1 | undefined; 0) | |

g) proc % outside = (file file) bool :
   (line ended (file) v page ended (file) v file ended (file));
10.5.3. Formatted transput

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

a) proc format = (file file, tamr of tamr) :
   (for p of file := 1; format of file :=
    collection list pack ("(" + file of tamr + ")", loc int := 1)) ;

b) proc % collection list pack = (string s, ref int p) string :
   (string t := collection (s, p);
    while e [p] = ",", do t +=: "," + collection (e, p); p +=: 1; t) ;

c) proc % collection = (string s, ref int p) string :
   (int n, q; string f := (p +=: 1; insertion (s, p)) ;
    q := p; replicator (s, p, n);
    e [p] = "(" | string t = collection list pack (s, p);
    to n do f +=: t | p := q; f +=: picture (s, p, loc [1 : 14] int));
    f + insertion (s, p)) ;

d) proc % insertion = (string s, ref int p) string :
   (int q = p; skip insertion (s, p); s [q := p +=: 1]) ;

e) proc % skip insertion = (string s, ref int p) :
   while (p > 1) 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, wtypik)) | p +=: 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 +=: s [p]; p +=: 1) ; n := (t = "" | t | string int ("+" + t, (10))) ;

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

i) proc % picture = (string format, ref int p, ref [] int frame) string :
   begin int n, int po = p; bool a;
   op ? = (string s) bool :
     (skip insertion (format, p); p > [ format | false | int q = p; replicator (format, p, n); a := q = p;
       (format [p] = "s" | p +=: 1);
       (char in string (format [p], loc int, s) | p +=: 1; true | p := q; false)) ;

   proc % intreal pattern = (ref [1 : 7] int frame) bool :
     (num mould (frame [2 : 4]) | frame [1] := 1; l);
     (? "n" | num mould (frame [3 : 5]) | frame [1] := 2; l);
     (? "a" | num mould (frame [6 : 7]) | frame [1] := 3; l);
     false. l: true);

   proc num mould = (ref [1 : 3] int frame) bool :
     (?: "nu" | frame [1] := n); (?: "nu" | frame [3] +=: n);
     (?: "nu" | frame [2] := 1); (?: "u" | frame [2] := 2);

   proc string mould = (ref [] int frame) bool : (?: "t" | true | while ? "a" do frame [4] +=: n; format [p] = "," v
     format [p] = ",") ;

   for i to 14 do frame [1] := 0; frame [8] := 10;
   (intreal pattern (frame [1 : 7]) | (? "n" |
     frame [1] := 2; intreal pattern (frame [8 : 14])) ; end) ;
10.5.3. continued

\[\text{(string mould (frame) | frame [1] := (frame [4] = I ∧ a | 9 | 6));}\]
\[\text{end; (if b | frame [1] := 8 | ?(a | frame [1] := 8 | frame [1] := 0; end);}\]
\[\text{(format p) := "(" | while ? "," do skip lit (format, p); p += 1);}\]
\[\text{end; skip insertion (format, p); format [po : p - 1]}\]
\[\text{end;}\]

(In connection with 10.5.3.1 see Table III.)

10.5.3.1. Formatted output

a) \text{proc out = (file file, tamref, tamref, [1] outtype x):} \text{(format (file, tamref), out (file, x)) ;} \\

b) \text{proc out = (file file, [1] : outtype x):} \\
\text{begin string format = format of file; ref int p = forp of file; for k to [x do} \\
\text{[I] := format = format of file; ref int q, j := 0; \[I : [I] int frame; \text{rep; j += 1;}}\\
\text{step: while (do insertion (file, format, p) p > [false | format (p) = ",") do p += 1; \text{[j := [y | end);}}\\
\text{p > [false | format end of file [p := 1]; step);}}\\
\text{q := p; picture (format, q, frame);}}\\
\text{[frame [1] int, real, real, compl, compl, string, intoh, boolean, char);}}\\
\text{int : (\{L int t; (t := y [j] |}}\\
\text{edit L int (file, t, format, p, frame); rep));\})\} incomp;}}\\
\text{real : (\{L real z; (z := y [j] |}}\\
\text{edit L real (file, z, format, p, frame); rep));\})\} incomp;}}\\
\text{compl : (\{L compl u; (u := y [j] |}}\\
\text{edit L compl (file, u, format, p, frame); rep));\})\} incomp;}}\\
\text{string : (\{L files := 0 files\} char s; (s := y [j] | frame [4] = 0 |}}\\
\text{put (file, s) | edit string (file, s [I, format, p, frame); rep));}}\\
\text{char : (char a; (a := y [j] |}}\\
\text{edit string (file, a, format, p, frame); rep));\})\} incomp;}}\\
\text{intoh : (int i; (i := y [j] |}}\\
\text{edit choice (file, i, format, p); rep));\})\} incomp;}}\\
\text{bool : (bool b; (b := y [j] | edit bool (file, b, format, p); rep));}}\\
\text{incomp : (value error of file | rep | put (file, y [j]; undefined);}}\\
\text{end: skip;}}\\
\text{end;}}\\
\text{c) proc \% edit L int = (file f, L int t, string format, ref int p, \[I int fr :} \\
\text{(string s = L int string (fr [4] + 1, fr [2]);}}\\
\text{(s = \"\" | \"\" value error of f | put (f, t); undefined |}}\\
\text{edit string (f, s, format, p, fr));\})\}}
d) proc % edit & real = (file f, L real x, string format, ref int p, int fr) :  
    (string s = stringed L real (x, fr); int t := -1;  
    (= char in string ("10", t, s) | char in string ("e", t, s));  
    (t = [ s | (= value error of f | put (f, L); undefined) |  
    edit string (f, s, format, p, fr))));  

e) proc % stringed L real = [L real x, [1 int fr] string :  

f) proc % edit L compl = (file f, L compl s, string format, ref int p,  
    [1 int fr]):  
    edit string (f, ([1 : 14] int g := fr; g [1] := 1; stringed L real  
    (re s, g [1 : ?]) + "" + stringed L real (im s, g [8 : 14]))),  
    format, p, fr);  

g) proc % edit string = (file f, string s, format, ref int p, [1 int fr]) :  
    begin int p1 := 1, n; bool supp; string s := x;  
    op ? = (char s) bool :  
    (do insertion (f, format, p); p > [ format |  
    false | int q := p; replator (format, p, n);  
    (supp := format [p] = "n" | p + := 1);  
    (= char in string (format [p], loc int, s) | p + := 1; true |  
    p := q; false));  
    proc copy = : (((= supp | put string (f, s [p1])); p1 + := 1);  
    proc intreal mould = : (? "n"; sign mould (fr [3]); int mould;  
    (? "n" | copy; int mould | s [p1] = ";" | p1 + := 1);  
    (? "n" | copy; sign mould (fr [6]); int mould));  
    proc sign mould = : (int sign) ; (sign = 0 |  
    (s [p1] = ";" | (= value error of file | undefined) |  
    s [p1] := (s [p1] = ";" | (sign = "n", "n") | "n");  
    (? "n" | sign mould zero (s, p1, p1 + n | n := 0);  
    to n + 1 do copy; p + := 1);  
    proc int mould = : (l: (? "n" | boolean as := true; to n do  
    (s [p1] = "0" & as | put string (file, "">")); p1 + := 1)  
    as := false; copy); l); (? "d" | to n do copy; l));  
    proc string mould = : while ? "a" do to n do copy;  
    fr [1] > 3 | p + := 1; copy; intreal mould)  
    end ;  

h) proc % edit choice = (file f, int c, string format, ref int p) :  
    (c > 0 | do insertion (f, format, p); p + := 1; to c - 1 do  
    (skip lit (format, p); format [p] = "," | p + := 1 | undefined);  
    do lit (f, format, p);  
    while format [p] = ",") do (p + := 1; skip lit (format, p)); p + := 1 |  
    undefined);  

i) proc % edit bool = (file f, bool b, string format, ref int p) :  
    (do insertion (f, format, p); (format [p + 1] = "," | p + := 1);  
    (b | do lit (f, format, p); p + := 1; skip lit (format, p) |  
    skip lit (format, p); p + := 1; do lit (f, format, p)) |  
    put string (f, (b | "n" | ",")); p + := 1);  

j) proc % do insertion = (file f, string s, ref int p) :  
    while (p > [ s | false | do align (f, s, p) | true |  
    do lit (f, s, p)) do skip ;
10.5.3.1. continued 2

k) proc % do align = (file f, string s, ref int p) bool :
    (int q = p; int n; replocator (s, p, n);
    {s [p] = 'n' | 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] = 'n' | to n do new line (f); l |
    s [p] = 'k' | char of bfile of f := n; l; p := q; false.
    l: p += 1; true);

l) proc % do lit = (file f, string s, ref int p) bool :
    (int q = p; int n; replocator (s, p, n); {s [p] = 'n' | while (a [p += 1] = 'n' | s [p += 1] = 'n' | true) do
    put string (f, s [p]); true | p := q; false)) ;

10.5.3.2. Formatted input

a) proc inf = (file file, tomof tomof, [] inttype w) :
    {format (file, tomof); in (file, w));

b) proc in = (file file, [1 :] inttype w) :
    begin string format = format of file; ref int p = forp of file;
    for k to 1 do
    (I :) ref simplint y = straightint [k]; int q, j := 0;
    [1 : I] int frame;
    rep: j += 1;
    step: while (exp insertion (file, format, p); p > [ format | false | format [p] = ",") do p += 1; (j > [ y | end));
    (p > [ format | (-format end of file | p := 1); step);
    q := p; picture (format, q, frame);
    (frame [I] int, real, real, compl, compl, string, intth, bool, char);
    int: (ref l int ii; (ii := y [j] |
    indid l int (file, ii, format, p, frame); rep)))) ; incomp;
    real: (ref l real xx; (xx := y [j] |
    indid l real (file, xx, format, p, frame); rep)))) ; incomp;
    compl: (ref l compl ss; (ss := y [j] |
    indid l compl (file, ss, format, p, frame); rep)))) ; incomp;
    string: (ref string ss; string t; (as := y [j] |
    (frame [I] = 0 | get (file, as) |
    indid string (file, t, format, p, frame); as[0 1] := t); rep));
    char: (ref char ca; string t; (as := y [j] |
    indid string (file, t, format, p, frame);
    (ref char : ca) := t [I]; rep) | incomp);
    intth: (ref int ti; (ti := y [j] |
    indid choice (file, ti, format, p); rep)) ; incomp;
    bool: (ref bool; (bb := y [j] |
    indid bool (file, bb, format, p); rep);)
    incomp: (value error of file | rep | undefined);
    end: skip
end ;

c) proc % indid l int = (file f, ref l int i, string format, ref int p, [1 int fr) :
    (string t; indid string (f, t, format, p, fr);
    i := l string int (t, fr [I]));

d) proc % indid l real = (file f, ref l real x, string format, ref int p, [1 int fr) :
    (string t; indid string (f, t, format, p, fr);
    x := l string int (t));
10.5.3.2. continued

e) proc % indit L compl = (file f, ref L compl s, string format, ref int p, [1 int fr]) : (string t; int i; indit string (f, t, format, p, fr); a := (char in string ("1", t, t); 1 string real (t [1 : i - 1]) & 1 string real (t [i+1:]));
f) proc % indit string = (file f, ref string s, string format, ref int p, [1 int fr]) : begin int n; bool supp; char k; string x; priority ! = 8;
    op ? = (char s) bool :
    (exp insertion (f, format, p); p > [ format | false | int q = p; repetitator (format, p, n);
        (supp := format [p] = "s" | p :== 1);  
        (char in string (format [p], loc int, s) | p +== 1; true | p := q; false));
    op ! = (char s, char o) char :
        char k := string (next loc int, s) | (supp | "" | k) |
            char supp := a; ((char error of f) supp | supp |
                undefined; o));
    op ! = (char s, a) char : (string s : o ! a;  
    proc char next = char : (get string (f, k); k);  
    proc intreal mould := : (? "n"; sign mould (fr [3]); int mould;
        (? "n" | x :== "x" | "" | "n"; int mould); |
        (? "n" | x :== "0" | "0"; sign mould (fr [3]; intmould)));  
    proc sign mould = (int sign) : (sign = 0 | x :== "n" | |
        int j := 0; (? "f" | n := 0;  
            for i to n + 1 while next = "n" do j := s; |
            x :== (k = "n" ^ k = "n" ^ sign = 1 | k | |
                (k = "a") | j := 1; backspace (f)); |
            for i from j + 1 to n + 1 do x :== "0123456789" | "0");  
            proc int mould = : (i: (? "n" | int j; |
                for i to n while next = "a" do j := i; backspace (f); |
                from j to n do x :== "0123456789" | "0"; I); |
                (? "a" | to n do x :== "0123456789" | "0"; I));  
            proc string mould := :  
                while ? "a" do to n do x := = (supp | "" | next;  
                    (fr [1] = 0 v fr [1] = 0) | string mould =: intreal mould; |
                    (fr [1] > 0 | "1" | "1" | intreal mould); i := x))
end;

g) proc % indit choices = (file f, ref int o, string format, ref int p) :
    (exp insertion (f, format, p); p +== 5; o := 1;  
    while ask lit (f, format, p) do  
        (o +== 1; format [p] = "n" | p +== 1 | undefined);  
    while format [p] = "n" do (p +== 1; skip lit (format, p));  
    p +== 1; exp insertion (f, format, p));
h) proc % indit bool = (file f, ref bool b, string format, ref int p) :
    (exp insertion (f, format, p); (format [p + 1] = "n" | p +== 2;  
        (b := ask lit (f, format, p) | p +== 1; skip lit (format, p) |  
            p +== 1; ask lit (f, format, p) undefined | char k;  
            get string (f, k); b := (k = "1" | true | k = "0" | false));  
    p +== 1; exp insertion (f, format, p));
i) proc % exp insertion = (file f, string s, ref int p) :
    while (p > 1 | s | false | do align (f, s, p) | true |  
        exp lit (f, s, p) do skip;  
    j) proc % exp lit = (file f, string s, ref int p) bool :
        (int q = p; int n; repetitator (s, p, n); (s [p] = "n" | int r = p;  
            to n do (p := r; while (s [p] +== 1) = "n" | s [p +== 1] = "n" |  
                true) do (char k; get string (f, k); k + s [p] undefined); true |  
                p := q; false));
10.5.3.2. continued 2

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

10.5.4. Binary transput

a) proc % to bin = (file f, simplout x) [] int :  
    a value of node 'row of integral' whose lower bound is one, and  
    whose upper bound depends on the value of 'f'; furthermore,  
    a = from bin (f, x, to bin (f, x)) a ;

b) proc % from bin = (file f, simplout v, [] int y) simplout :  
    a value, if one exists, of the node of the actual-parameter corre-  
    sponding to y, such that y = to bin (f, from bin (f, v, y)) a ;

{On some channels a more straightforward way of transput is available.  
Some properties of this binary transput depend on the particular implementa-
tion, others can be deduced from 10.5.4.1.

10.5.4.1. Binary output

a) proc write bin = ([] outtype x) : put bin (stand back, x) ;

b) proc put bin = (file file, [] outtype x) :  
    if bin possible [chan of file] ^ opened of file ^  
    put possible [chan of file]  
    then if − set possible [chan of file] then state def of file  
    then (state get of file v − state bin of file | undefined)  
    else state def of file := state bin of file := true;  
    state get of file := false  
    fi;
    for k to [ x do ([] ] simplout y = straightout x [k];  
    for j to [ y do ([] ] int bin = to bin (file, y [j]);  
    ref bfile b = bfile of file;  
    ref int p = page of b, l = line of b, a = char of b;  
    for i to [ bin do (next p (file); check p (file);  
    (book of b) (p, l, a) := bin [i]; a := i;  
    (p = lpage of b & l = lline of b |  
    (a > lchar of b | lchar of b := a) |  
    lpage of b := p; lline of b := l; lchar of b := a)))
    else undefined
    fi;

10.5.4.2. Binary input

a) proc read bin = ([] inttype x) : get bin (stand back, x) ;

b) proc get bin = (file file, [] inttype x) :  
    if bin possible [chan of file] ^ opened of file ^  
    get possible [chan of file]  
    then if − set possible [chan of file] then state def of file  
    then (−state get of file v − state bin of file | undefined)
10.5.4.2. continued

e else state def of file := state bin of file :=
    state get of file := true
fi;
for k to 1 do ((1 :) simpout y = straightin = [k];
    for j to 1 do ((1 : j) flex int bin := to bin (file, y [j]);
        ref file b = bfile of file;
        for i to 1 bin do (next plo (file); check plo (file);
            bin [i] := (book of b) [page of b, line of b, char of b];
            char of b ::= 1);
        (\ref L int ii; (ii ::= y [j] | (ref L int : ii) ::= from bin (file, ii, bin)));
        (\ref L real xx; (xx ::= y [j] | (ref L real : xx) ::= from bin (file, xx, bin)));
        (\ref L compl xx; (xx ::= y [j] | (ref L compl : xx) ::= from bin (file, xx, bin)));
        (\ref string ss; (ss ::= y [j] | (ref string : ss) ::= from bin (file, ss, bin)));
        (ref char oo; (oo ::= y [j] | (ref char : oo) ::= from bin (file, oo, bin)));
        (ref bool bb; (bb ::= y [j] | (ref bool : bb) ::= from bin (file, bb, bin))));
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.}

10.6. Standard postlude

a) look (stand in); look (stand out); look (stand back)
11. Examples

11.1. Complex square root

A declaration in which \texttt{compsqrt} is a procedure-with-[\texttt{complex}]parameter-[\texttt{complex}]-mode-identifier (here \texttt{[complex]} stands for structured-with-real-field-letter-r-letter-e-and-real-field-letter-i-letter-m.):

a) \texttt{proc compsqrt = (compl a) \# the square root whose real part is nonnegative of the complex number \(a\) \# compl :}

b) \texttt{begin real \(a = \text{re} \ a, y = \text{im} \ a;\)}

c) \texttt{real \(r = \text{sqrt} ((\text{abs} \ a + \text{sqrt} (\text{sqrt} (\text{abs} \ a + 2 + y + 2)) / 3));\)}

d) \texttt{real \(i = \text{sqrt} (y / (2 \times r));\)}

e) \texttt{(a \geq 0 | r \geq 0 | y \geq 0 | \text{abs} \ i \geq 0 | y \geq 0 | r \geq 0 | - r)}

\textbf{end}

\texttt{[complex]-calls (8.6.2) using compsqrt:}

g) \texttt{compsqrt (w)}

h) \texttt{compsqrt (- 3,14)}

i) \texttt{compsqrt (- 1)}

11.2. Innerproduct 1

A declaration in which \texttt{innerproduct 1} is a procedure-with-integral-parameter-and-procedure-with-integral-parameter-real-parameter-andprocedure-with-integral-parameter-real-parameter-real-mode-identifier:

a) \texttt{proc innerproduct 1 = (int n, proc (int) real \(x, y) \# real :}

\texttt{comment the innerproduct of two vectors, each with \(n\) components, \(x \langle t \rangle, y \langle t \rangle, t = 1, \ldots, n\), where \(x\) and \(y\) are arbitrary mappings from integer to real number comment}

b) \texttt{begin real \(s := \text{long} \ 0;\)}

c) \texttt{for \(i\) to \(n\) do \(s := s + \text{long} \ x \langle i \rangle \times \text{long} \ y \langle i \rangle;\)}

d) \texttt{short \(s\)}

e) \texttt{end}

\textbf{Real-calls (8.6.2) using innerproduct 1:}

g) \texttt{innerproduct 1 \(m, \langle \text{int} \ j \rangle \) real : \#1 \[j \], \(\langle \text{int} \ j \rangle \) real : \#1 \[j \]}
11.3. continued

Real-calls using innerproduct 3:

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

11.4. Innerproduct 3

A declaration in which innerproduct 3 is a procedure-with-reference-to-integral-parameter-and-integral-parameter-and-procedure-real-parameter-and-procedure-real-parameter-real-mode-identifier:

a) proc innerproduct 3 = (ref int i, int n, proc real x[i, y[i]) real:
   " 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 "
   b) begin long real s := long 0;
   c) for k to n do (i := k; s + := leng x[i] x leng y[i]);
   d) short s
   e) end

A real-call using innerproduct 3:

f) innerproduct 3 (j, 8, x1 [j], y1 [j + 1])

11.5. Largest element


a) proc abemax = (ref [1], 1) real a, result, ref real y,
   b) " subscripts " ref int i, k1 :
   " the absolute value of the element of greatest absolute value of the
   matrix a is assigned to y, and the subscripts of this element to i
   and k "
   c) begin y := -1;
   d) for p to 1 upb a do for q to 2 upb a do
   e) if abe a [p, q] > y then y := abe a [{i := p}, (k := q)] fi
   f) end

Void-calls using abemax:

g) abemax (x3, z, i, j)
h) abemax (x3, z, loc int, loc int)
11.6. Euler summation

a) \textit{proc euler} = (proc (int) real f, real eps, int tim) real : \\
\textit{\$\$ the sum for } i \textit{ from 1 to infinity of } f(i), \textit{ 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 \textit{ tim times in succession. This transformation is particularly efficient in the case of a slowly convergent or divergent alternating series } \$
\textit{b) begin int n := 1, t; real mn, ds := eps; [1 : 16] real m;}
\textit{c) real sum := (m [1] := f (1)) / 2;}
\textit{d) for } i \textit{ from 2 while } (t := (abs ds < eps \mid t + 1 \mid 1)) \leq \textit{tim do}
\textit{e) begin mn := f (i);}
\textit{f) for } k \textit{ to } n \textit{ do begin mn := ((ds := mn) + m [k]) / 2;}
\textit{g) n m [k] := ds end;}
\textit{h) sum plus (ds := (abs mn < abs m [n] \land n < 16 \mid n plus 1; m [n] := mn; mn / 2 \mid mn))}
\textit{j) end;}
\textit{k) sum}
\textit{end}

A call using \textit{euler}:

\textit{m) euler ((int i) real : (odd i \mid - 1 / i \mid 1 / i), 10^-5, 2)}

11.7. The norm of a vector

a) \textit{proc norm} = (ref [1:] real a) real : \\
\textit{\$ the euclidean norm of the vector } a \$
\textit{b) (long real a := long 0;}
\textit{c) for } k \textit{ to upb a do a plus lena [k] + 2;}
\textit{d) short long sqrt (abs a))}

For a use of \textit{norm} as a call, see 11.8.e.

11.8. Determinant of a matrix

a) \textit{proc det} = (ref [1 ; 1 :] real a; ref [1 : upb a] int p) real : \\
\textit{\$ the determinant of the square matrix } a \textit{ of order } n \textit{ by the method of Crout with row interchanges; } a \textit{ is replaced by its triangular decomposition } l \times u \textit{ with all } u [k, k] = 1. \textit{The vector } p \textit{ gives as output the pivotal row indices; the } k\text{-th pivot is chosen in the } k\text{-th column of } l \textit{ such that } abs l [i, k] / \textit{row norm is maximal. } \$
\textit{b) if upb a = 2 upb a}
\textit{c) then int n = upb a;}
\textit{d) for } i \textit{ to } n \textit{ do v [1] := norm (a [i]);}
\textit{e) for } k \textit{ to } n \textit{ do}
\textit{g) begin int k1 = k - 1; ref int pk = p [k];}
\textit{h) ref [1] real al = a [1 ; k1]; au = a [1 : k1];}
\textit{i) ref [1] real ak = a [k], ka = a [1 ; k];}
\textit{j) alk = al [k], kaau = au [1 ; k];}
11.8. continued

```
k)  for i from k to n do
l)      begin ref real atk = ka [i];
m)      if (s := abs (atk := innerproduct 2 (al [i], kau)) / v [i]) > r
n)      then r := s; pk := i fi
o)      end;
p)      v [pk] := v [k]; pivot := ka [pk]; ref [] real apk = a [pk];
q)      for j to n do
r)      begin ref real akj = ak [j], apkj = apk [j];
s)      r := akj; akj := if j < k then apkj
t)      else (apkj - innerproduct 2 (alk, au [, j])) / pivot fi;
u)      if pk < k then apkj := - r fi
v)      end;
w)      d := pivot
x)      end;
y)      d
z)      fi
```

A call using det:

aa) det (y2, i1)

11.9. Greatest common divisor

a) proc gcd = (int a, b) int :

> the greatest common divisor of two integers

> (b = 0 | abs a | gcd (b, a :: b))

A call using gcd:

b) gcd (n, 124)

11.10. Continued fraction

a) op / = ([1 ::] real a; [1 :: upb a] real b) real :

> comment the value of a / b is that of the continued fraction

> \( \frac{a_1}{b_1 + \frac{a_2}{b_2 + \cdots + \frac{a_n}{b_n}}} \) comment

b) (upb a = 0 | 0 | a [1] / (b [1] + a [2 ::] / b [2 ::]))

A formula using /:

c) x1 / y1

(The use of recursion may often be elegant rather than efficient as in
the recursive procedure 11.9 and the recursive operation 11.10. See, how-
ever, 11.11 and 11.14 for examples 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, so = 5;
h) heap const zero, one; value of zero := 0; value of one := 1;
i) op = = (form a, ref const b)
    bool : (ref const eo; (eo ::= a | eo ::= 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 v b = zero | zero | a := one | b := zero | a | triple := (a, times, b));
m) op /= (form a, b) form : (a = zero v ¬(b = zero) | zero |
    b := one | a | triple := (a, by, b));

op += (form a, ref const b) form : (a = one v (b := zero) | one |
    b :=: one | a | triple := (a, to, b));
o) proc derivative of = (form e, with respect to ref var x) form :
p) begin ref const eo; ref var ev; ref triple et; ref call ef;
q) case ev, et, ef ::= e in
r)  \phi ev \phi (ev ::= z | one | zero),
s)  \phi et \phi begin form u = left operand of et, v = right operand of et;
t)  form udash = derivative of (u, \phi with respect to \phi x);
u)  udash = derivative of (v, \phi with respect to \phi x);
v)  case operator of et in
w)  udash + vdash, udash - vdash,
x)  u * vdash + udash * v, (udash - et * vdash) / v,
y)  (eo ::= v | u * v)
    (heap const a; value of a ::= value of eo - 1;a) * udash)
    esaa
z) \phi ef \phi begin ref function f = function name of ef;
aa) form g = parameter of ef;
b) ref var y = bound var of f;
c) heap function fdash := (y, derivative of (body of f, y));
d) (call := (fdash, g1)) * derivative of (g, x)
e) end

aa) out \phi eo \phi zero
    esaa \phi ev, et, ef, eo \phi
    end \phi derivative \phi ;
af) proc value of = (form e) real :
ag) begin ref const eo; ref var ev; ref triple et; ref call ef;
aah) case eo, ev, et, ef ::= e in
ai) \phi eo \phi value of eo,
aj) \phi ev \phi value of ev,
ak) \phi et \phi begin real u = value of (left operand of et),
al) v = value of (right operand of et);
am) case operator of et in
an) u + v, u - v, u * v, u / v, exp (v * ln (u)) esaa
   end ;
11.11. continued

a0) \[ \text{def} \text{ def function } f = \text{function name of } \text{ef}; \]

a1) \[ \text{value of bound var of } f := \text{value of (parameter of } \text{ef)}; \]

aq) \[ \text{value of (body of } f) \]

end \[ \text{esac} \text{ def } \text{ea, ev, et, ef } \]

end \[ \text{value of } \text{e}; \]

ar) \[ \text{heap form } f, g; \text{ heap var } a := ("a", \infty), b := ("b", \infty), x := ("x", \infty); \]

as) \[ \text{start here: read } ((\text{value of } a, \text{value of } b, \text{value of } x)); \]

st) \[ f := a + x / (b + x); g := (f + one) / (f - one); \]

au) \[ \text{print } ((\text{value of } a, \text{value of } b, \text{value of } x, \text{value of (derivative of } g, \text{with respect to } x))); \]

end \[ \text{example of formula manipulation } \]

11.12. Information retrieval

a) \[ \text{begin mode } ra = \text{ref auth, rb = ref book,} \]

b) \[ \text{struct auth = (string name, ra next, rb book),} \]

c) \[ \text{book = (string title, rb next);} \]

d) \[ \text{ra auth, first auth := nil, last auth; rb book;} \]

f) \[ \text{string name, title; int i; file input, output;} \]

g) \[ \text{open (input, remote in); open (output, remote out);} \]

h) \[ \text{outf (output, } \text{&} \)

i) \[ \text{"to enter a new author, type " author", a space, and his name."} \]

j) \[ \text{"to enter a new book, type " book", a space, the name of the author, a new line, and the title."} \]

k) \[ \text{"for a listing of the books by an author, type " list", a space, and his name."} \]

l) \[ \text{"to find the author of a book, type " find", a new line, and the title."} \]

m) \[ \text{"to end type " end" "al", "."}; \]

n) \[ \text{PROC update =} \]

o) \[ \text{if (ra : first auth) := nil;} \]

p) \[ \text{then auth := first auth := last auth := auth := (name, \infty, \infty);} \]

q) \[ \text{else auth := first auth; while (ra : auth) \#; nil do} \]

r) \[ \text{(name = name of auth \& known | auth := next of auth);} \]

s) \[ \text{last auth := next of last auth := auth := auth := (name, \infty, \infty);} \]

r) \[ \text{known: skip} \]

fi \[ \text{end declaration prelude sequence } \]

s) \[ \text{client: inf (input, } \text{fo("author" "book", "list", "find", "end", "")}, \text{al, al, \& i));} \]

t) \[ \text{case i in author, publ, list, find, end, error esac;} \]

u) \[ \text{author: in (input, name); update; client;} \]

v) \[ \text{publ: in (input, (name, title)); update;} \]

w) \[ \text{if (rb : book of auth) := nil;} \]

x) \[ \text{then book of auth := book := (title, \infty);} \]

y) \[ \text{else book := book of auth; while (rb : next of book) \#; nil do} \]

z) \[ \text{(title \& title of book \& client \& book := next of book);} \]

aa) \[ \text{fi; client;} \]
11.12. continued

ab) list: in (input, name); update;
sa)  outf (output, $p"author:"$30a$, name);
sc)  f (rb : book of auth) := nil
sd)  then outf (output, "no publications")
se)  else while (rb : book) #: nil do
sf)  begin if line number (output) = max line [remote out]
sg)  then outf (output, $41k"continued on next page"p

sh)                          "author:"$30a41k"continued":$1", name)
ai)                           outf (output, $30a$, title of book);
aj)  book := next of book
ak)  end
al) fi; client;

am) find: in (input, (loc string, title)); auth := first auth;
bn) while (ra : auth) #: nil do
bo) begin book := book of auth;
bp) while (rb : book) #: nil do
bq) if title = title of book
br) then outf (output, "$l"author:"$30a$, name of auth);
bs) client
bt) else book := next of book
bu) fi; auth := next of auth
bv) end;
aw) outf (output, "$l"unknown":$1", client;
ax) end: put (output, (new page, "signed off", close));
ay) close (input).

11.13. Cooperating sequential processes

b) begin int nmb magazine slots, nmb producers, nmb consumers;
   read ((nmb magazine slots, nmb producers, nmb consumers));
c) [1 : nmb producers] file infile, [1 : nmb consumers] file outfile;
d) for i to nmb producers do open (infile[i], inchannel[i]);
   \* inchannel and outchannel are defined in a surrounding range \*
   for i to nmb consumers do open (outfile[i], outchannel[i]);
   mode page = [1 : 80, 1 : 133] char;
   [1 : nmb magazine slots] ref page magazine;
   int * pointers of a cyclic magazine \* index := 1, exedex := 1;
   sema full slots := 0, free slots := nmb magazine slots;
   in buffer busy := 1, out buffer busy := 1;
   proc par call = (proc (int) p, int n) \* calls n incarnations of p in
   parallel \* (n > 0 | par (p (n), par call (p, n - 1)));
   proc producer = (int i) : do (heap page page; get (infile[i], page);
   + free slots; + in buffer busy;
   o) magazine [index] := page; index := nmb magazine slots +:= 1;
   p) + full slots; + in buffer busy;
   q) proc consumer = (int i) : do (page page;
   r) + full slots; + out buffer busy;
   s) page := magazine [exindex]; exindex := nmb magazine slots +:= 1;
   t) + free slots; + out buffer busy; put (outfile[i], page);
   u) par (par call (producer, nmb producers),
   par call (consumer, nmb consumers))
   end
11.14. Towers of Hanoi

a) begin proc p = (int ma, de, ma) :
b) (ma > 0 | p (me, 6 - me - de, ma - 1));
c) out (stand out, (me, de, ma));
   † move from peg 'me' to peg 'de' piece 'ma' †
d) p (6 - me - de, de, ma - 1));
e) for k to 8 do (outf (stand out, "$k_0=","d_1,
   n((2\times k+15)\times 16)(2(2(\times 3(d)\times x)\times x)\times x)\times x)\times x)\times x)\times k);
f) end

12. Glossary

12.1. Technical terms

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., "assign", "assigned", "assignment") is given once, usually as infinitive (e.g., "assign").

action 2.2, 2.2.5
ALGOL 68 program 4.4
apostrophe 1.1.6.c
applied occurrence 4.1.2.a
appoint 6.0.2.a
a priori value 5.1.0.2.b
arithmetic value 2.2.3.1.a
assign 2.2.2.1, 8.3.1.2.c
asterisk 1.1.2.a
automaton 1.1.1.a
backfile 10.5.1.aa,cc
balance 6, 6.4.1
blind alley 1.1.2.d
bold-face shift 3.1.2.b
case clause 9.4.c,d
channel 10.5.1.aa,bb
character 2.2.3.1.a,f
close a file 10.5.1.ii
collateral 2.2.5.a, 6.2.2.a
colon 1.1.2.a
comma 1.1.2.a
compatible 5.5.1.dd,nn
compile 2.3.c
component of 2.2.2.h
composite 3.1.2.d
complete 5.0.2.a
computer 1.1.1.a
conformity case clause 9.4.g
constant 5
constituent 1.1.6.f
contain 1.1.6.b
conversion key 5.5.1.ff
copy 2.2.4.1.a
create a file 10.5.1.gg
defining occurrence 2.2.2.c, 4.1.2.a
denote 1.1.6.c
deproceduring 8.2, 8.2.2
dereferencing 8.2, 8.2.1
descendent 1.1.6.e,f
describe 2.2.3.3.b
descriptor 2.2.3.3.a
develop 7.1.2.c
direct constituent 1.1.6.e,f
direct descendent 1.1.6.e
direct production 1.1.2.c
divided by 2.2.3.1.c
edit 5.5.1.11
elaborate collateral 6.2.2.a
elaboration 1.1.6.h, 6.0.2.a
element 2.2.2.k
end of file 10.5.1.cc
English language 1.1.1.b
envelope 1.1.6.j
environment enquiry 10.1
equivalent to 2.2.2.h
establish a file 10.5.1.gg
expect 5.5.1.gg
extended language 1.1.1.a
extension 1.1.7
external object 2.2.1
false 2.2.3.1.e
field 2.2.2.k
file 5.5.1.aa,10.5.1,10.5.1.ee
firmly coerced from 4.4.3.a
firm position 8.2
follow 1.1.6.a
formal language 1.1.1.b
format 2.2.3,2.2.3.4,5.5
halt 6.0.2.a
hardware language 1.1.8.b
heap 8.5.1
hipping 8.2,8.2.7
hold 2.2
home 4.1.2.b
human being 1.1.1.a
hypernotion 1.3
hyphen 1.1.6.c.iv
identification string 10.5.1.cc
identify 2.2.2.b,c
implementation 2.3.c
index 2.2.3.3.a
indication-applied occurrence 4.2.2.a
indication-defining occurrence
indinit 5.5.1.mm,2.2.2.c,4.2.2.a
initiate 2.2.2.g,6.0.2.a
input 5.5.1.aa,10.5
inseparable 2.2.5.a
instance 2.2.1
integer 2.2.3.1.a,b,c,d
integral equivalent 2.2.3.1.f
internal object 2.2.1
interrupt 6.0.2.a
12.1. continued

in the reach of 4.4.2.c
in the sense of numerical analysis 2.2.3.1.c
large syntactic marks 1.1.2.a
layout procedure 10.5.2.1
length number 2.2.3.1.b
light-face shift 3.1.2.b
list of metanotions 1.1.3.c
list of notions 1.1.2.c
literal 5
lock a file 10.5.1.ji
loosely related 4.4.3.c
lower bound 2.2.3.3.b
lower state 2.2.3.3.b
meaningful (program) 4.4
member 1.1.2.d
metalanguage 1.1.3.a
metamember 1.1.3.d
metanotion 1.1.3.a
minus 2.2.3.1.c
mode 1.1.6.i, 2.2.4.1.a
multiple value 2.2.3, 2.2.3.3
name 2.2.2.1, 2.2.3.5
nil 2.2.2.1, 2.2.3.5.a
notion 1.1.2.a
object 2.2, 2.2.1
object program 2.3.c
occurrence 1.1.6.d, 2.2.1
offset 2.2.3.3.b
offspring 1.1.6.e
of the same mode as 2.2.2.h
open a file 10.5.1.ff
operator-applied occurrence 4.3.2.a
operator-defining occurrence 2.2.2.c,
original 1.1.6.c /4.3.2.a
other syntactic marks 1.1.2.a
output 5.5.1.aa, 10.5
overflow 6.0.2.b
paranotion 1.1.6.c
pass on 6
permanent 2.2.2.a
plain value 2.2.3, 2.2.3.1
point 1.1.2.a
portrayal 2.2.4.1.d
position of the file 5.5.1.ff
possess 2.2.2.d
possibly intended (program) 2.3.c
pragmatic remark 1.3
precede 1.1.6.a
preelaboration 1.1.6.i
precend 1.1.6.i
prescope 1.1.6.i
present 5.5.1.ff
prevalue 1.1.6.i
proceduring 8.2, 8.2.3
production 1.1.2.e
production rule 1.1.2.a
production tree 1.1.6.e
productive 1.1.2.d
proper (program) 4.4
protect 6.0.2.d, 7.1.2.b
protonotion 1.1.2.b
publication language 1.1.8.b
quintuple 2.2.3.3.b
random access 10.5.1.bb
reach 4.4.2.a, 4.4.2.c
read 5.5.1.ji
real number 2.2.3.1.a,b,c,d
refer to 2.2.2.h
related 4.4.3.b
relationship 2.2, 2.2.2
repetitive statement 9.3
representation 1.1.8.a
representation language 1.1.1.a
reset a file 10.5.1.hh
resume 5.0.2.a
routine 2.2.2.f, 2.2.3.4
rowing 8.2, 8.2.6
scope 1.1.6.i, 2.2.3.5.a, 2.2.4.2
scratch a file 10.5.1.hh
select 2.2.3.2, 2.2.3.3.a
semantics 1.1.2.a, 2.3
semicolon 1.1.2.a
sequential access 10.5.1.bb
serial 2.2.5.a
set a file 10.5.1.oo
shield 4.4.4.a
show 4.4.4.b
smaller than 2.2.2.h
small syntactic marks 1.1.2.a
soft position 8.2
sort 6
standard declaration 10.a
standard file 10.5.1.3
standard mathematical constant 10.3
standard mathematical function 10.3
standard operation 10.2
standard priority 10.2.0
straightening 5.5.1.dd, 10.5.0.2
strict language 1.1.1.a
stride 2.2.3.3.b
string 5.3.2
strongly coerced from 4.4.3.a
strong position 8.2
structured value 2.2.3, 2.2.3.2
structured from 2.2.4.1.d
subvalue 2.2.2.k
successor 6.0.2.a
supersede 8.3.1.2.a
suppress 5.5.1.11
symbol 1.1.2.b
synchronization operation 10.4
12.2. Paranoations

Given below are the indicators of the rules yielding production rules for the originals of the given paranoations and other protonations or giving instructive appearances of, or representations for, the given symbols. Ordinary type font without hyphens is used in order to shorten the text by using hyphens in a conventional way.

absolute value of symbol 3.1.1.c  character frame 5.5.5.b
action token 3.0.4.a  - pattern 5.5.5.a
actual declarator 7.1.1.c,d,e,l,o,p,  characters to bytes symbol 3.1.1.c
- declarer 7.1.1.b  character = 3.1.1.d
- lower bound 7.1.1.t  - token 3.0.9.d
- parameter 7.4.1.b  choice clause 6.4.1.c,d
- row of rover 7.1.1.r  clause 6.1.1.a, 6.2.1.b,c,d,f,
- upper bound 7.1.1.t  6.3.1.a, 6.4.1.a, 8.1.1.a
adic indication 4.2.1.g  - train 6.1.1.b
alignment 5.5.1.i  closed clause 6.3.1.a
and symbol 3.1.1.c  close symbol 3.1.1.e
assignment 8.3.1.1.a  coerced 8.2.0.1.a
at symbol 3.1.1.e  cohesion 8.5.0.1.a
balance 6.2.1.e  collateral clause 6.2.1.b,c,d,f
base 8.6.0.1.a,b  - declaration 6.2.1.a
basic token 3.0.1.a  collection 5.5.1.b
begin symbol 3.1.1.e  comma symbol 3.1.1.b
binal - 3.1.1.c  comment 3.0.9.b
bits denotation 5.2.1.a  - item 3.0.9.c
- symbol 3.1.1.d, 10.2.8.a  - symbol 3.1.1.i
boolean choice mould 5.5.4.b  completer 6.1.1.1
- denotation 5.1.3.1.a  completion symbol 3.1.1.f
- pattern 5.5.4.a  complex frame 5.5.6.c
booleans to bits symbol 3.1.1.c  - pattern 5.5.6.a
boundscript 8.6.1.1.l  - symbol 3.1.1.d, 10.2.7.a
bus symbol 3.1.1.e  condition 6.4.1.b
by - 3.1.1.h  conditional clause 6.4.1.a
bytes - 3.1.1.d, 10.2.9.a  conformity relation 8.3.2.1.a
call 8.6.2.1.a  - relation 8.3.2.1.b
caption 7.5.1.b  conforms to and becomes symbol
cast 8.3.4.1.a  3.1.1.c
- of symbol 3.1.1.c  - symbol 3.1.1.c
chain 3.0.1.c  confrontation 8.3.0.1.a
character denotation 5.1.4.1.a  - token 3.0.4.d
- joint 3.1.1.c  conjugate of symbol 3.1.1.c

(Denn eben, wo Begriffe fehlen,
Da stellt ein Wort zur rechten Zeit sich ein.
Faust,  J.W. von Goethe.)
12.2. continued

constant 6.0.1.d
declaration 6.2.1.a, 7.0.1.a
- prelude 6.1.1.b
- token 3.0.5.a
declarer 7.1.1.a
denotation 5.0.1.a
- token 3.0.3.a
destination 8.3.1.1.b
differs from symbol 3.1.1.c
digit eight 3.0.3.d
- symbol 3.1.1.b
- five 3.0.3.d
- symbol 3.1.1.b
- four 3.0.3.d
- symbol 3.1.1.b
- frame 5.5.2.e
- nine 3.0.3.d
- symbol 3.1.1.b
- one 3.0.3.d
- symbol 3.1.1.b
- seven 3.0.3.d
- symbol 3.1.1.b
- six 3.0.3.d
- symbol 3.1.1.b
- three 3.0.3.d
- symbol 3.1.1.b
token 3.0.3.c
two 3.0.3.d
- symbol 3.1.1.b
- zero 3.0.3.d
- symbol 3.1.1.b
divided by and becomes symbol 3.1.1.c
- symbol 3.1.1.c
do symbol 3.1.1.h
down - 3.1.1.c
dyadic formula 8.4.1.h
- indicant 1.1.5.b
- indication 4.2.1.d
- operator 4.5.1.d
dynamic replication 5.5.1.h
either symbol 3.1.1.d
ext else clause 6.4.1.e
- if symbol 3.1.1.h
- symbol 3.1.1.e, 9.4
end symbol 3.1.1.e
equals - 3.1.1.c
exit 2.1.e
exponent frame 5.5.3.f
- part 5.1.2.1.g
expression 6.0.1.b
extras token 3.0.9.a
false symbol 3.1.1.b
field declarator 7.1.1.g
- selector 7.1.1.i
file symbol 3.1.1.d, 10.5.1.2

fi symbol 3.1.1.e, 9.4
flexible - 3.1.1.d
flip flop 3.0.3.e
flip symbol 3.1.1.b
floating point mould 5.5.3.d
- numeral 5.1.2.1.e
flop symbol 3.1.1.b
formal declarator 7.1.1.c,d,e,m,n,o,
- declarer 7.1.1.b
- lower bound 7.1.1.v
- parameter 5.4.1.e
- row of rover 7.1.1.r
- upper bound 7.1.1.v
format denotation 5.5.1.a
- symbol 3.1.1.d
formatter - 3.1.1.b
formula 8.4.1.a
for symbol 3.1.1.h
fractional part 5.1.2.1.d
frame 5.5.1.r
from symbol 3.1.1.h
generator 8.5.1.1.a
global - 8.5.1.1.c
gomma 5.4.1.d

go on symbol 3.1.1.f
to - 3.1.1.f
heap - 3.1.1.d
hip token 3.0.8.a
identifier 4.1.1.a
identity declaration 7.4.1.a
relation 8.3.3.1.a
relator 8.3.3.1.b
if symbol 3.1.1.e, 9.4
imaginary part of - 3.1.1.c
indexer 8.6.1.1.k
indicant 1.1.5.b
indication 4.2.1.a
insert 5.5.1.e
insertion 5.5.1.d
integral choice pattern 5.5.2.f
- denotation 5.1.1.1.a
- mould 5.5.2.d
- part 5.1.2.1.c
- pattern 5.5.2.a
- symbol 3.1.1.d
is at least - 3.1.1.c
- most - 3.1.1.c
- greater than - 3.1.1.c
- less - - 3.1.1.c
- not - 3.1.1.c
- symbol 3.1.1.c
label 6.1.1.k
- identifier 4.1.1.b
- symbol 3.1.1.e
lengthen - 3.1.1.c
letter a 3.0.2.b
  - symbol 3.1.1.a
  - aleph 3.0.2.b
  - b 3.0.2.b
  - symbol 3.1.1.a
  - c 3.0.2.b
  - symbol 3.1.1.a
  - d 3.0.2.b
  - symbol 3.1.1.a
  - e 3.0.2.b
  - symbol 3.1.1.a
  - f 3.0.2.b
  - symbol 3.1.1.a
  - g 3.0.2.b
  - symbol 3.1.1.a
  - h 3.0.2.b
  - symbol 3.1.1.a
  - i 3.0.2.b
  - symbol 3.1.1.a
  - j 3.0.2.b
  - symbol 3.1.1.a
  - k 3.0.2.b
  - symbol 3.1.1.a
  - l 3.0.2.b
  - symbol 3.1.1.a
  - m 3.0.2.b
  - symbol 3.1.1.a
  - n 3.0.2.b
  - symbol 3.1.1.a
  - o 3.0.2.b
  - symbol 3.1.1.a
  - p 3.0.2.b
  - symbol 3.1.1.a
  - q 3.0.2.b
  - symbol 3.1.1.a
  - r 3.0.2.b
  - symbol 3.1.1.a
  - s 3.0.2.b
  - symbol 3.1.1.a
  - t 3.0.2.b
  - token 3.0.2.a
  - t symbol 3.1.1.a
  - u 3.0.2.b
  - symbol 3.1.1.a
  - v 3.0.2.b
  - symbol 3.1.1.a
  - w 3.0.2.b
  - symbol 3.1.1.a
  - x 3.0.2.b
  - symbol 3.1.1.a
  - y 3.0.2.b
  - symbol 3.1.1.a
  - z 3.0.2.b
  - symbol 3.1.1.a
library postlude 2.1.f
library prelude 2.1.c
list 3.0.1.d
  - proper 3.0.1.g
  - separator 3.0.1.e
literal 5.5.1.j
local generator 8.5.1.1.b
  - symbol 3.1.1.d
long denotation 5.1.0.1.b
  - symbol 3.1.1.d
loose replicatable suppressible
  - character frame 5.5.1.m
  - = digit - 5.5.1.m
  - = zero - 5.5.1.m
  - suppressible character - 5.5.1.m
  - = complex - 5.5.1.m
  - = exponent - 5.5.1.m
  - = point - 5.5.1.m
lower bound of symbol 3.1.1.c
  - state = 3.1.1.c
minus and becomes - 3.1.1.c
  - symbol 3.1.1.c
mode declaration 7.2.1.a
  - identifier 4.1.1.b
  - indication 4.2.1.b
  - standard 4.2.1.c
  - symbol 3.1.1.d
modulo and becomes symbol 3.1.1.c
  - symbol 3.1.1.c
monadic formula 8.4.1.g
  - indicant 1.1.5.b
  - indication 4.2.1.f
  - operand 8.4.1.f
  - operator 4.3.1.e
new lower bound 8.6.1.1.h
  - = part 8.6.1.1.g
nil symbol 3.1.1.g
  - not - 3.1.1.c
number token 3.0.3.b
odd symbol 3.1.1.c
of = 3.1.1.e
one token 7.3.1.b
  - plus one = 7.3.1.c
  - plus one = 7.3.1.d
  - plus one - 7.3.1.e
  - plus one - 7.3.1.f
  - plus one - 7.3.1.g
  - plus one - 7.3.1.h
  - plus one - 7.3.1.i
  - plus one - 7.3.1.j
open symbol 3.1.1.e
operand 8.4.1.c
operation declaration 7.5.1.a
operation symbol 3.1.1.d
operation 4.3.1.a,b,c
    - token 3.0.4.b
option 3.0.1.b
or symbol 3.1.1.c
over and becomes - 3.1.1.c
    - symbol 3.1.1.c
pack 3.0.1.h
package 3.0.1.i
parallel symbol 3.1.1.e
parameters pack 7.1.1.bb
particular program 2.1.d
phrase 6.0.1.a
picture 5.5.1.c
plain denotation 5.1.0.1.a
plus and becomes symbol 3.1.1.c
    - 1 times - 3.1.1.c
plus-minus 3.0.4.c
plus symbol 3.1.1.c
point frame 5.5.3.c
    - symbol 3.1.1.b
power of ten 5.1.2.1.i
primary 8.1.1.d
priority declaration 7.3.1.a
    - one indication 4.2.1.e
    - + plus one - 4.2.1.e
    - - - - plus one - 4.2.1.e
    - - - - - plus one - 4.2.1.e
    - - - - - - plus one - 4.2.1.e
    - - - - - - - plus one - 4.2.1.e
    - - - - - - - - plus one - 4.2.1.e
priority one operator 4.3.1.b
    - + plus one - 4.3.1.b
    - - plus one - 4.3.1.b
    - - - plus one - 4.3.1.b
    - - - - plus one - 4.3.1.b
    - - - - - plus one - 4.3.1.b
    - - - - - - plus one - 4.3.1.b
    - - - - - - - plus one - 4.3.1.b
    - - - - - - - - plus one - 4.3.1.b
    - - - - - - - - - plus one - 4.3.1.b
    - - - - - - - - - - plus one - 4.3.1.b
    - - - - - - - - - - - plus one - 4.3.1.b
quote image 5.1.4.1.c
    - symbol 3.1.1.i
radix 5.5.2.c
    - mould 5.5.2.b
range 4.1.1.e
real denotation 5.1.2.1.a
    - mould 5.5.3.b
    - part of symbol 3.1.1.c
    - pattern 5.5.3.a
    - symbol 3.1.1.d
reference to - 3.1.1.d
replicatable suppressible character
    - frame 5.5.1.n
    - digit - 5.5.1.n
    - zero - 5.5.1.n
replicated literal 5.5.1.k
replication 5.5.1.g
replicator 5.5.1.f
representation of symbol 3.1.1.c
round - 3.1.1.c
routine denotation 5.4.1.a
row of character - 5.3.1.b
    - - pattern 5.5.7.b
    - display 6.0.1.h
secondary 8.1.1.c
selection 8.5.2.1.a
sensa symbol 3.1.1.d, 10.4.a,b,c,d
sequence 3.0.1.d
    - proper 3.0.1.g
    - sequencer 6.1.1.j
sequence separator 3.0.1.f
sequencing token 3.0.7.a
serial clause 6.1.1.a
shorten symbol 3.1.1.c
sign frame 5.5.1.p
mould 5.5.1.l
    - symbol 3.1.1.c
single declaration 6.1.1.d
skip symbol 3.1.1.g
slice 8.6.1.1.a
source 8.3.1.1.c
space symbol 3.1.1.b
special token 3.0.10.a
stagnant mould 5.5.3.e
part 5.1.2.1.f
standard postlude 2.1.g
    - prelude 2.1.b
    - statement 6.0.1.c
    - interlude 6.1.1.i
    - prelude 6.1.1.c
strict lower bound 7.1.1.u
    - upper - 7.1.1.u
string denotation 5.3.1.a
    - frame 5.5.7.c
    - item 5.1.4.1.b
string pattern 5.5.7.a
  - symbol 3.1.1.d, 10.2.10.a
structure 6.2.1.g,h
  - display 6.0.1.g
  - symbol 3.1.1.d
subscript 8.6.1.1.i
sub symbol 3.1.1.e
suite of clause trains 6.1.1.f,g
suppressible character frame 5.5.1.q
  - complex - 5.5.1.q
  - digit - 5.5.1.q
  - exponent - 5.5.1.q
  - point - 5.5.1.q
syntactic token 3.0.6.a
tertiary 8.1.1.b
th element of symbol 3.1.1.c
then clause 6.4.1.e
  - if symbol 3.1.1.h
  - symbol 3.1.1.e, 9.4
times and becomes symbol 3.1.1.c
  - ten to the power choice 5.1.2.1.h
  - symbol 3.1.1.b
  - symbol 3.1.1.e
to - 3.1.1.h
transform 5.5.8.1.a
trimmer 8.6.1.1.f
trimscript 8.6.1.1.j
true symbol 3.1.1.b
union of - 3.1.1.d
unit 6.1.1.e
unitary clause 8.1.1.a
  - declaration 7.0.1.a
upper bound of symbol 3.1.1.c
  - state - - 3.1.1.c
  - up - 3.1.1.c
  - to - 3.1.1.e
variable 6.0.1.e
  - point numeral 5.1.2.1.b
virtual declarator 7.1.1.c,d,e,l,o,p,
  - declarer 7.1.1.b
  - lower bound 7.1.1.s
  - parameter 7.1.1.y
  - row of rower 7.1.1.r
  - upper bound 7.1.1.s
  - void declarer 7.1.1.z
while symbol 3.1.1.h
zero frame 5.5.1.o