# Revised Report on the Algorithmic Language

# Algol 68

Edited by

A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck,
C. H. A. Koster, M. Sintzoff, C. H. Lindsey,
L. G. L. T. Meertens and R. G. Fisker

This Edition, which is issued as a Supplement to ALGOL Bulletin number 47, includes all errata authorised by the ALGOL 68 Support subcommittee of IFIP WG2.1 up to the end of 1978.

CONTENTS



PART I

Preliminary definitions

## PART II

## Fundamental constructions

## PART III

## Context dependence



## PART IV

## Elaboration-independent constructions

PART V

Environment and examples

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 Seegmueller [6], and Aad van Wijngaarden [9]. In [6] and [8], the descriptional 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 subsequent meetings between April 1966 and December 1968, held in Kootwijk near Amsterdam, Warsaw, Zandvoort near Amsterdam, Tirrenia near Pisa and North Berwick near Edinburgh, a number of successive approximations to a final report, commencing with [10] and followed by a series numbered MR 88, MR 92, MR 93, MR 95, MR 99 and MR 100, were submitted by a team working in Amsterdam, consisting first of A. van Wijngaarden and Barry Mailloux, later reinforced by John Peck, and finally by Kees Koster. Versions were used during courses on the language held at various centres, and the experience gained in explaining the language to skilled audiences and the reactions of the students influenced the succeeding versions. The final version, MR 101 [11], was adopted by the Working Group on December 20th 1968 in Munich, and was subsequently approved for publication by the General Assembly of I.F.I.P. Since that time, it has been published in Numerische Mathematik [12], and translations have been made into Russian [13], into German [14], into French [15], and into Bulgarian [16]. An "Informal Introduction", for the benefit of the uninitiated reader, was also prepared at the request of the Working Group [18].

The original authors acknowledged with pleasure and thanks the wholehearted 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. Sintzoff, P. Branquart, J. Lewi and P. Wodon for numerous brainstorms, A.J.M. van Gils of Apeldoorn, G. Goos and his group at Munich, also for [7], G.S. Tseytin 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.

Since the publication of the Original Report, much discussion has taken place in the Working Group concerning the further development of the language. This has been influenced by the experience of many people who saw disadvantages in the original proposals and suggested revised or extended features. Amongst these must be mentioned especially: I.R. Currie, Susan G. Bond, J.D. Morison and D. Jenkins of Malvern (see in [17]), in whose dialect of ALGOL 68 many features of this Revision may already be found; P. Branquart, J.P. Cardinael and J. Lewi of Brussels, who exposed many weaknesses (see in [17]); Ursula Hill, H. Woessner and H. Scheidig of Munich, who discovered some unpleasant consequences; the contributors to the Rapport d'Evaluation [19]; and the many people who served on the Working Group subcommittees on Maintenance and Improvements (convened by M. Sintzoff) and on Transput (convened by C.H. Lindsey). During the later stages of the revision, much helpful advice was given by H. Boom of Edmonton, W. Freeman of York, W.J. Hansen of Vancouver, Mary Zosel of Livermore, N. Yoneda of Tokyo, M. Rain of Trondheim, L. Ammeraal, D. Grune, H. van Vliet and R. van Vliet of Amsterdam, G. van der Mey of Delft, and A.A. Baehrs and A.F. Rar of Novosibirsk. The editors of this revision also wish to acknowledge that the wholehearted cooperation, support, interest, criticism and violent objections on the part of the members of WG 2.1 have continued unabated during this time.

[1] J.V. Garwick, J.M. Merner, P.Z. Ingerman and M. Paul, Report of the ALGOL-X - I-O Subcommittee, WG 2.1 Working Paper, July 1966.
[2] C.A.R. Hoare, Record Handling, WG 2.1 Working Paper, October 1965; also AB.21.3.6, November 1965.
[3] P. Naur (Editor), Revised Report on the Algorithmic Language ALGOL 60, Regnecentralen, Copenhagen, 1962, and elsewhere.
[4] P. Naur, Proposals for a new language, AB.18.3.9, October 1964.
[5] P. Naur, Proposals for introduction on aims, WG 2.1 Working Paper, October 1965.
[6] G. Seegmueller, A proposal for a basis for a Report on a Successor to ALGOL 60, Bavarian Acad. Sci., Munich, October 1965.
[7] G. Goos, H. Scheidig, G. Seegmueller and H. Walther, Another proposal for ALGOL 67, Bavarian Acad. Sci., Munich, May 1967.
[8] N. Wirth, A Proposal for a Report on a Successor of ALGOL 60, Mathematisch Centrum, Amsterdam, MR 75, October 1965.
[9] A. van Wijngaarden, Orthogonal Design and Description of a Formal Language, Mathematisch Centrum, Amsterdam, MR 76, October 1965.
[10] A. van Wijngaarden and B.J. Mailloux, A Draft Proposal for the Algorithmic Language ALGOL X, WG 2.1 Working Paper, October 1966.
[11] A. van Wijngaarden (Editor), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Report on the Algorithmic Language ALGOL 68, Mathematisch Centrum, Amsterdam, MR 101, February 1969.
[12] *idem*, Numerische Mathematik, Vol. 14, pp. 79-218, 1969.

[13] Soobshchenie ob algoritmicheskom yazyke ALGOL 68, translation into Russian by A.A. Baehrs, A.P. Ershov, L.L. Zmievskaya and A.F. Rar, Kybernetica, Kiev, Part 6 of 1969 and Part 1 of 1970.
[14] Bericht ueber die Algorithmische Sprache ALGOL 68, translation into German by I.O. Kerner, Akademie-Verlag, Berlin, 1972.
[15] Définition du Langage Algorithmique ALGOL 68, translation into French by J. Buffet, P. Arnal, A. Quéré (Eds.), Hermann, Paris, 1972.
[16] Algoritmichniyat yezik ALGOL 68, translation into Bulgarian by D. Toshkov and St. Buchvarov, Nauka i Yzkustvo, Sofia, 1971.
[17] J.E.L. Peck (Ed.), ALGOL 68 Implementation (proceedings of the I.F.I.P. working conference held in Munich in 1970), North Holland Publishing Company, 1971.
[18] C.H. Lindsey and S.G. van der Meulen, Informal introduction to ALGOL 68, North Holland Publishing Company, 1971.
[19] J.C. Boussard and J.J. Duby (Eds.), Rapport d'Evaluation ALGOL 68, Revue d'Informatique et de Recherche Opérationelle, B2, Paris, 1970.

## 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) This present Revision of the language is made in response to the directive of the parent committee, I.F.I.P. TC 2, to the Working Group to "keep continually under review experience obtained as a consequence of this (original) publication, so that it may institute such corrections and revisions to the Report as become desirable". In deciding to bring forward this Revision at the present time, the Working Group has tried to keep in balance the need to accumulate the maximum amount of experience of the problems which arose in the language originally defined, as opposed to the needs of the many teams at present engaged in implementation, for whom an early and simple resolution of those problems is imperative.

d) Although the language as now revised differs in many ways from that defined originally, no attempt has been made to introduce extensive new features and, it is believed, the revised language is still clearly recognizable as "ALGOL 68". The Working Group has decided that this present revision should be "the final definition of the language ALGOL 68", and the hope is expressed that it will be possible for implementations at present in preparation to be brought into line with this standard.

e) The Working Group may, from time to time, define sublanguages and extended capabilities, by means of Addenda to this Report, but these will always be built on the language here defined as a firm foundation. Moreover, variants more in conformity with natural languages other than English may be developed. To coordinate these activities, and to maintain contact with implementers and users, a Subcommittee on ALGOL 68 Support has been established by the Working Group.

f) The members of the Group, influenced by several years of experience with ALGOL 60 and other programming languages, have always 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 formalized two-level grammar, with the semantics expressed in natural language, but making use of some carefully and precisely defined terms and concepts. It is recognized, however, that this method may be difficult for the uninitiated reader.

0.1.2. Orthogonal design

The number of independent primitive concepts has been 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 while trying to avoid deleterious superfluities.

0.1.3. Security

ALGOL 68 has been designed in such a way that most 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., 11.7.

0.1.4.1. Static mode checking

The syntax of ALGOL 68 is such that no mode checking during run time is necessary, except when the programmer declares a **UNITED-variable** and then, in a **conformity-clause**, explicitly demands a check on its mode.

0.1.4.2. 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, it can be determined in a finite number of steps whether an arbitrary given sequence of symbols is a program.

0.1.4.3. 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 independent compilation, specification of the mode of all nonlocal quantities is provided; see the remarks after 2.2.2.c.

0.1.4.4. 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.

0.1.4.5. 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 insight into the essential, fundamental concepts of computing and a new description technique.

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

0.2.1. Values in ALGOL 68

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

b) Each plain value is either arithmetic, i.e., of **'integral'** or **'real'** mode and then it is of one of several sizes, or it is of **'boolean'** or **'character'** or **'void'** mode. Machine words, considered as sequences of bits or of bytes, may also be handled.

c) In ALGOL 60, values can be composed 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 approximately to the ALGOL 60 string; examples of structured values are complex numbers and symbolic formulae.

d) In ALGOL 68 the concept of a "name" is introduced, i.e., a value which is said to "refer to" another value; such a name-value pair corresponds to the ALGOL 60 variable. However, a 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.

f) In contrast with plain values, the significance of a name or routine is, in general, dependent upon the existence of the storage cells referred to or accessed. Therefore, the use of names and routines is subject to some restrictions related to their "scope". However, the syntax of ALGOL 68 is such that in many cases the check on scope can be made at compile time, including all cases where no use is made of features whose expressive power transcends that of ALGOL 60.

### 0.2.2. Declarations in ALGOL 68

a) Whereas ALGOL 60 has type declarations, array declarations, switch declarations and procedure declarations, ALGOL 68 features the **identity-declaration** whose expressive power includes all of these, and more. The **identity-declaration**, although theoretically sufficient in itself, is augmented by the **variable-declaration** for the convenience of the user.

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

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

0.2.3. Dynamic storage allocation in ALGOL 68

Whereas ALGOL 60 (apart from "***own*** dynamic arrays") implies a "stack"-oriented storage-allocation regime, sufficient to cope with objects having nested lifetimes (an object created before another object being guaranteed not to become inaccessible before that second one), ALGOL 68 provides, in addition, the ability to create and manipulate objects whose lifetimes are not so restricted. This ability implies the use of an additional area of storage, the "heap", in which garbage-collection techniques must be used.

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 latter facility is conducive to more efficient object programs under many circumstances, since it allows discretion to the implementer to choose, in many cases, the order of elaboration of certain constructs or even, in some cases, whether they are to be elaborated at all. Thus the user who expects his "side effects" to take place in any well determined manner will receive no support from this Report. 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 "transput" declarations, which make it possible, at run time, to obtain data from and to deliver results to external media.

0.2.6. Some particular constructions in ALGOL 68

a) The ALGOL 60 concepts of block, compound statement and parenthesized expression are unified in ALGOL 68 into the **serial-clause**. A **serial-clause** may be an **expression** and yield 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 yield a value.

b) The ALGOL 60 concept of subscripting 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 **row-displays** and **structure-displays**, which serve to compose the multiple and structured values mentioned in 0.2.1.c from other, simpler, values.

d) The ALGOL 60 for statement is modified into a more concise and efficient **loop-clause**.

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**, and a **conformity-clause**, which allows a selection depending upon the actual mode of a value.

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.

### 0.3. Comparison with the language defined in 1968

The more significant changes to the language are indicated in the sections which follow. The revised language will be described in a new edition of the "Informal Introduction to ALGOL 68" by C.H. Lindsey and S.G. van der Meulen, which accompanied the original Report.

#### 0.3.1. Casts and routine texts

Routines without parameters used to be constructed out of a **cast** in which the **cast-of-symbol** (:) appeared. This construct is now one of the forms of the new **routine-text**, which provides for procedures both with and without parameters. A new form of the **cast** has been provided which may be used in contexts previously not possible. Moreover, both **void-casts** and **procedure-PARAMETY-yielding-void-routine-texts** must now contain an explicit **void-symbol**.

#### 0.3.2. Extended ranges

The new **range** which is established by the **enquiry-clause** of a **choice-clause** (which encompasses the old **conditional-** and **case-clauses**) or of a **while-part** now extends into the controlled **serial-clause** or **do-part**.

#### 0.3.3. Conformity clauses

The **conformity-relation** and the **case-conformity** which was obtained by extension from it are now replaced by a new **conformity-clause**, which is a further example of the **choice-clause** mentioned above.

0.3.4. Modes of multiple values

A new class of modes is introduced, for multiple values whose elements are themselves multiple values. Thus one may now write the **declarer** [ ] ***string***.

Moreover, multiple values no longer have "states" to distinguish their flexibility. Instead, flexibility is now a property of those names which refer to multiple values whose size may change, such names having distinctive modes of the form '**reference to flexible ROWS of MODE**'.

0.3.5. Identification of operators

Not only may two **operators**, related to each other by the modes of their operands, not be declared in the same **range**, as before, but now, if two such **operators** be declared in different reaches, any attempt to identify from the inner reach the one in the outer reach will fail. This gives some benefit to the implementer and removes a source of possible confusion to the user.

0.3.6. Representations

The manner in which **symbols** for newly defined **mode-indications** and **operators** are to be represented is now more closely defined. Thus it is clear that the implementer is to provide a special alphabet of bold-faced, or "stropped", marks from which **symbols** such as ***person*** may be made, and it is also clear that **operators** such as >> are to be allowed.

0.3.7. Standard prelude

In order to ease the problems of implementers who might wish to provide variants of the language suitable for environments where English is not spoken, there are no longer any **field-selectors** known to the user in the **standard-prelude**, with the exception of *re* and *im* of the mode ***compl***. The **identifiers** and other **indicators** declared in the **standard-prelude** could, of course, easily be defined again in some **library-prelude**, but this would not have been possible in the case of **field-selectors**.

0.3.8. Line length in transput

The lines (and the pages also) of the "book" used during transput may now, at the discretion of the implementer, be of varying lengths. This models more closely the actual behaviour of most operating systems and of devices such as teleprinters and paper-tape readers.

0.3.9. Internal transput

The transput routines, in addition to sending data to or from external media, may now be associated with **row-of-character-variables** declared by the user.

0.3.10. Elaboration of formats

The dynamic **replicators** contained in **format-texts** are now elaborated as and when they are encountered during the formatted transput process. This should give an effect more natural to the user, and is easier to implement.

0.3.11. Features removed

Certain features, such as proceduring, **gommas** and formal bounds, have not been included in the revision.

0.4. Changes in the method of description

In response to the directive from the Working Group "to make its study easier for the uninitiated reader", the Editors of this revision have rewritten the original Report almost entirely, using the same basic descriptional technique, but applying it in new ways. It is their hope that less "initiation" will now be necessary.

The more significant changes in the descriptional technique are indicated below.

0.4.1. Two-level grammar

a) While the syntax is still described by a two-level grammar of the type now widely known by the name "Van Wijngaarden", new techniques for using such grammars have been applied. In particular, the entire identification process is now described in the syntax using the metanotion "**NEST**", whose terminal metaproductions are capable of describing, and of passing on to the descendent constructs, all the declared information which is available at any particular node of the production tree.

b) In addition, extensive use is made of "predicates". These are notions which are deliberately made to yield blind alleys when certain conditions are not met, and which yield empty terminal productions otherwise. They have enabled the number of syntax rules to be reduced in many cases, while at the same time making the grammar easier to follow by reducing the number of places where a continuation of a given rule might be found.

c) It has thus been possible to remove all the "context conditions" contained in the original Report.

0.4.2. Modes

a) In the original Report, modes were protonotions of possibly infinite length. It was assumed that, knowing how an infinite mode had been obtained, it was decidable whether or not it was the same as some other infinite mode. However, counterexamples have come to light where this

was not so. Therefore, it has been decided to remove all infinities from the process of producing a finite **program** and, indeed, this can now be done in a finite number of moves.

b) A mode, essentially, has to represent a potentially infinite tree. To describe it as a protonotion of finite length requires the use of markers ('**MU definition**'s) and pointers back to those markers ('**MU application**'s) within the protonotion. However, a given infinite tree can be "spelled" in many ways by this method, and therefore a mode becomes an equivalence class comprised of all those equivalent spellings of that tree. The equivalence is defined in the syntax using the predicates mentioned earlier.

### 0.4.3. Extensions

The need for many of the extensions given in the original Report had been removed by language changes. Some of the remainder had been a considerable source of confusion and surprises. The opportunity has therefore been taken to remove the extension as a descriptional mechanism, all the former extensions now being specified directly in the syntax.

### 0.4.4. Semantics

a) In order to remove some rather repetitious phrases from the semantics, certain technical terms have been revised and others introduced. The grammar, instead of producing a terminal production directly, now does so by way of a production tree. The semantics is explained in terms of production trees. Paranotions, which designate constructs, may now contain metanotions and "hypernotions" have been introduced in order to designate protonotions.

b) A model of the hypothetical computer much more closely related to a real one has been introduced. The elaboration of each construct is now presumed to take place in an "environ" and, when a new **range** is entered (and, in particular, when a routine is called), a new "locale" is added to the environ. The locale corresponds to the new **range** and, if recursive procedure calls arise, then there exist many locales corresponding to one same routine. This supersedes the method of "textual substitution" used before, and one consequence of this is that the concept of "protection" is no longer required.

c) The concept of an "instance" of a value is no longer used. This simplifies certain portions of the semantics where, formerly, a "new instance" had to be taken, the effects of which were not always clear to see.

0.4.5. Translations

The original Report has been translated into various natural languages. The translators were not always able to adhere strictly to the descriptional method, and so the opportunity has been taken to define more clearly and more liberally certain descriptional features which caused difficulties (see 1.1.5).

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

# PART I

## Preliminary definitions

## 1. Language and metalanguage

### 1.1. The method of description

#### 1.1.1. Introduction

a) ALGOL 68 is a language in which algorithms may be formulated for computers, i.e., for automata or for human beings. It is defined by this Report in four stages, the "syntax" {b}, the "semantics" {c}, the "representations" {d} and the "standard environment" {e}.

b) The syntax is a mechanism whereby all the constructs of the language may be produced. This mechanism proceeds as follows:

(i) A set of "hyper-rules" and a set of "metaproduction rules" are given {1.1.3.4, 1.1.3.3}, from which "production rules" may be derived:

(ii) A "construct in the strict language" is a "production tree" {1.1.3.2.f} which may be produced by the application of a subset of those production rules; this production tree contains static information {i.e., information known at "compile time"} concerning that construct: it is composed of a hierarchy of descendent production trees, terminating at the lowest level in the "**symbols**"; with each production tree is associated a "nest" of properties, declared in the levels above, which is passed on to the nests of its descendents;

(iii) A "**program** in the strict language" is a production tree for the notion '**program**' {2.2.1.a}. It must also satisfy the "environment condition" {10.1.2}.

c) The semantics ascribes a "meaning" {2.1.4.1.a} to each construct {i.e., to each production tree} by defining the effect (which may, however, be "undefined") of its "elaboration" {2.1.4.1}. This proceeds as follows:

(i) A dynamic {i.e., run-time} tree of active "actions" is set up {2.1.4}; typically, an action is the elaboration of some production tree T in an "environ" consistent with the nest of T, and it may bring about the elaboration of descendents of T in suitable newly created descendent environs;

(ii) The meaning of a **program** in the strict language is the effect of its elaboration in the empty "primal environ".

d) A **program** in the strict language must be represented in some "representation language" {9.3.a} chosen by the implementer. In most cases this will be the official "reference language".

(i) A **program** in a representation language is obtained by replacing the **symbols** of a **program** in the strict language by certain typographical marks {9.3}.

(ii) Even the reference language allows considerable discretion to the implementer {9.4.a,b,c}. A restricted form of the reference language in which such freedom has not been exercised may be termed the "canonical form" of the language, and it is expected that this form will be used in algorithms intended for publication.

(iii) The meaning of a **program** in a representation language is the same as that of the **program** {in the strict language} from which it was obtained.

e) An algorithm is expressed by means of a **particular-program**, which is considered to be embedded, together with the standard environment, in a **program-text** {10.1.1.a}. The meaning of a **particular-program** {in the strict or a representation language} is the meaning of the **program** "akin" to that **program-text** {10.1.2.a}.

### 1.1.2. Pragmatics

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

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

{Some of the pragmatic remarks contain examples written in the reference language. In these examples, **applied-indicators** occur out of context from their **defining-indicators**. Unless otherwise specified, these occurrences identify those in the **standard-** or **particular-preludes** and the

**particular-postlude** (10.2, 10.3, 10.5) (e.g., see 10.2.3.12.a for *pi*, 10.5.1.b for *random* and 10.5.2.a for *stop*), or those in:

```
int i, j, k, m, n; real a, b, x, y; bool p, q, overflow; char c; format f;
    bytes r; string s; bits t; compl w, z; ref real xx, yy; union (int, real) uir;
    proc void task1, task2;
[1 : n] real x1, y1; flex [1 : n | real a1; [1 : m, 1 : n | real x2;
    [1 : n, 1 : n] real y2; [1 : n | int i1; [1 : m, 1 : n | int i2;
    [1 : n] compl z1;
proc x or y = ref real : if random < .5 then x else y fi;
proc ncos = (int i) real : cos (2 × pi × i / n);
proc nsin = (int i) real : sin (2 × pi × i / n);
proc finish = void : go to stop;
mode book = struct (string text, ref book next); book draft;
princeton: grenoble: st pierre de chartreuse: kootwijk: warsaw:
    zandvoort: amsterdam: tirrenia: north berwick: munich:
    finish.)
```

1.1.3. The syntax of the strict language

1.1.3.1. Protonotions

a) In the definition of the syntax of the strict language, a formal grammar is used in which certain syntactic marks appear. These may be classified as follows:
(i) "small syntactic marks", written, in this Report, as
"**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**", "**(**", "**)**";
(ii) "large syntactic marks", written, in this Report, as
"**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**", "**0**", "**1**", "**2**", "**3**", "**4**", "**5**", "**6**", "**7**", "**8**", "**9**";
(iii) "other syntactic marks", written, in this Report, as
"**.**" ("point"), "**,**" ("comma"), "**:**" ("colon"), "**;**" ("semicolon"), "**'**" ("apostrophe"), "**-**" ("hyphen") and "*****" ("asterisk").

{Note that these marks are in another type font than that of the marks in this sentence.}

b) A "protonotion" is a possibly empty sequence of small syntactic marks.

c) A "notion" is a {nonempty} protonotion for which a production rule can be derived {1.1.3.2.a, 1.1.3.4.d}.

d) A "metanotion" is a {nonempty} sequence of large syntactic marks for which a metaproduction rule is given or created {1.1.3.3.a}.

e) A "hypernotion" is a possibly empty sequence each of whose elements either is a small syntactic mark or is a metanotion.

{Thus the class of protonotions (b) is a subclass of the class of hypernotions. Hypernotions are used in metaproduction rules (1.1.3.3), in hyper-rules (1.1.3.4), as paranotions (1.1.4.2) and, in their own right, to "designate" certain classes of protonotions (1.1.4.1).}

{A "paranotion" is a hypernotion to which certain special conventions and interpretations apply, as detailed in 1.1.4.2.}

f) A "symbol" is a protonotion ending with **'symbol'**. {Note that the paranotion **symbol** (9.1.1.h) designates a particular occurrence of such a protonotion.}

{Examples:

b) **'variable point'**
c) **'variable point numeral'** (8.1.2.1.b)
d) "**INTREAL**" (1.2.1.C)
e) **'reference to INTREAL'**
f) **'letter a symbol'** .

Note that the protonotion **'twas brillig and the slithy toves'** is neither a symbol nor a notion, in that it does not end with **'symbol'** and no production rule can be derived for it. Likewise, "**LEWIS**" and "**CARROLL**" are not metanotions in that no metaproduction rules are given for them.}

g) In order to distinguish the various usages in the text of this Report of the terms defined above, the following conventions are adopted:

(i) No distinguishing marks {quotes, apostrophes or hyphens} are used in production rules, metaproduction rules or hyper-rules;

(ii) Metanotions, and hypernotions which stand for themselves {i.e., which do not designate protonotions}, are enclosed in quotes;

(iii) Paranotions are not enclosed in anything {but, as an aid to the reader, are provided with hyphens where, otherwise, they would have been provided with blanks};

(iv) All other hypernotions {including protonotions} not covered above are enclosed in apostrophes {in order to indicate that they designate some protonotion, as defined in 1.1.4.1.a};

(v) Typographical display features, such as blank space, hyphen, and change to a new line or new page, are of no significance (but see also 9.4.d).

{Examples:

(i) **LEAP :: local ; heap ; primal.** is a metaproduction rule;

(ii) "**INTREAL**" is a metanotion and designates nothing but itself;

(iii) **reference-to-INTREAL-identifier**, which is not enclosed in apostrophes but is provided with hyphens, is a paranotion designating a construct (1.1.4.2.a);

(iv) **'variable point'** is both a hypernotion and a protonotion; regarded as a hypernotion, it designates itself regarded as a protonotion;

(v) **'reference to real'** means the same as **'referencetoreal'**.}

1.1.3.2. Production rules and production trees

a) The {derived} "production rules" {b} of the language are those production rules which can be derived from the "hyper-rules" {1.1.3.4}, together with those specified informally in 8.1.4.1.d and 9.2.1.d.

b) A "production rule" consists of the following items, in order:
an optional asterisk ;
a nonempty protonotion N ;
a colon ;
a nonempty sequence of "alternatives" separated by semicolons ;
a point.
It is said to be a production rule "for" {the notion (1.1.3.1.c)} N.

{The optional asterisk, if present, signifies that the notion is not used in other production rules, but is provided to facilitate discussion in the semantics. It also signifies that that notion may be used as an "abstraction" (1.1.4.2.b) of one of its alternatives.}

c) An "alternative" is a nonempty sequence of "members" separated by commas.

d) A "member" is either
(i) a notion {and may then be said to be productive, or nonterminal},
(ii) a symbol {which is terminal},
(iii) empty, or
(iv) some other protonotion {for which no production rule can be derived}, which is then said to be a "blind alley".

{For example, the member **'reference to real denotation'** (derived from the hyper-rule 8.0.1.a) is a blind alley.}

{Examples:

b) **exponent part : times ten to the power choice,**
**power of ten.** (8.1.2.1.g) •
**times ten to the power choice :**
**times ten to the power symbol ;**
**letter e symbol.** (8.1.2.1.h)
c) **times ten to the power choice, power of ten** •
**times ten to the power symbol** •
**letter e symbol**
d) **times ten to the power choice** •
**power of ten** •
**times ten to the power symbol** •
**letter e symbol** }

e) A "construct in the strict language" is any "production tree" {f} that may be "produced" from a production rule of the language.

f) A "production tree" T for a notion N, which is termed the "original" of T, is "produced" as follows:

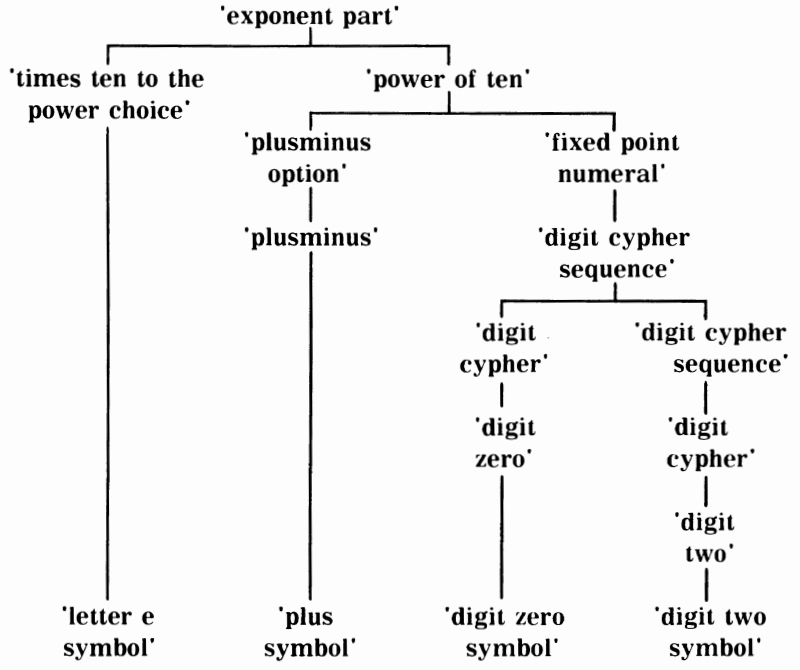
- let P be some {derived} production rule for N;
- a copy is taken of N;
- a sequence of production trees, the "direct descendents" of T, one produced for each nonempty member of some alternative A of P, is attached to the copy; the order of the sequence is the order of those members within A;
- the copy of the original, together with the attached direct descendents, comprise the production tree T.

A "production tree" for a symbol consists of a copy of that symbol {i.e., it consists of a **symbol**}.

The "terminal production" of a production tree T is a sequence consisting of the terminal productions of the direct descendents of T, taken in order.

The "terminal production" of a production tree consisting only of a **symbol** is that **symbol**.

{Example:




}

{The terminal production of this tree is the sequence of **symbols** at the bottom of the tree. In the reference language, its representation would be *e+02*.}

A "terminal production" of a notion is the terminal production of some production tree for that notion {thus there are many other terminal productions of **'exponent part'** besides the one shown}.

{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 some notion is so by virtue of a unique production tree, or by a set of production trees which differ only in such a way that the result of their elaboration is the same (e.g., production trees derived from rules 3.2.1.e (balancing), 1.3.1.d,e (predicates) and 6.7.1.a,b (choice of spelling of the mode of a **coercend** to be voided); see also 2.2.2.a).

Therefore, in practice, terminal productions (or representations thereof) are used, in this Report and elsewhere, in place of production trees. Nevertheless, it is really the production trees in terms of which the elaboration of programs is defined by the semantics of this Report, which is concerned with explaining the meaning of those constructs whose originals are the notion **'program'**.}

g) A production tree P is a "descendent" of a production tree Q if it is a direct descendent {f} either of Q or of a descendent of Q. Q is said to "contain" its descendents and those descendents are said to be "smaller" than Q. {For example, the production tree

**'plusminus option'**
|
**'plusminus'**
|
**'plus symbol'**

occurs as a descendent in (and is contained within and is smaller than) the production tree for **'exponent part'** given above.}

h) A "visible" ("invisible") production tree is one whose terminal production is not (is) empty.

i) A descendent {g} U of a production tree T is "before" ("after") a descendent V of T if the terminal production {f} of U is before (after) that of V in the terminal production of T. The {partial} ordering of the descendents of T thus defined is termed the "textual order". {In the example production tree for **'exponent part'** (f), the production tree whose original is **'plusminus'** is before that whose original is **'digit two'**.}

j) A descendent A of a production tree "follows" ("precedes") another descendent B in some textual order if A is after (before) B in that textual order, and there exists no visible {h} descendent C which comes between A and B. {Thus "immediately" following (preceding) is implied.}

k) A production tree A is "akin" to a production tree B if the terminal production {f} of A is identical to the terminal production of B.

1.1.3.3. Metaproduction rules and simple substitution

{The metaproduction rules of the language form a set of context-free grammars defining a "metalanguage".}

a) The "metaproduction rules" {b} of the language are those given in the sections of this Report whose heading begins with "Syntax", "Metasyntax" or "Metaproduction rules", together with those obtained as follows:

- for each given metaproduction rule, whose metanotion is M say, additional rules are created each of which consists of a copy of M followed by one of the large syntactic marks "**0**", "**1**", "**2**", "**3**", "**4**", "**5**", "**6**", "**7**", "**8**" or "**9**", followed by two colons, another copy of that M and a point.

{Thus, the metaproduction rule "**MODE1 :: MODE.**" is to be added.}

b) A "metaproduction rule" consists of the following items, in order:
an optional asterisk ;
a nonempty sequence M of large syntactic marks ;
two colons ;
a nonempty sequence of hypernotions {1.1.3.1.e} separated by semicolons ;
a point.

It is said to be a metaproduction rule "for" {the metanotion (1.1.3.1.d)} M.

{The asterisk, if present, signifies that the metanotion is not used in other metaproduction rules or in hyper-rules, but is provided to facilitate discussion in the semantics.}

{Examples:

**INTREAL :: SIZETY integral ; SIZETY real.** (1.2.1.C) •
**SIZETY :: long LONGSETY ; short SHORTSETY ; EMPTY.** (1.2.1.D)}

c) A "terminal metaproduction" of a metanotion M is any protonotion which is a "simple substitute" {d} for one of the hypernotions {on the right hand side} of the metaproduction rule for M.

d) A protonotion P is a "simple substitute" for a hypernotion H if a copy of H can be transformed into a copy of P by replacing each metanotion M in the copy by some terminal metaproduction of M.

{Thus two possible terminal metaproductions (c) of "**INTREAL**" are '**integral**' and '**long long real**'. This is because the hypernotions '**SIZETY integral**' and '**SIZETY real**' (the hypernotions of the metaproduction rule for "**INTREAL**") may, upon simple substitution (d), give rise to '**integral**' and '**long long real**', which, in turn, is because ' ' (the empty protonotion) and '**long long**' are terminal metaproductions of "**SIZETY**".}

{The metanotions used in this Report have been so chosen that no concatenation of one or more of them gives the same sequence of large syntactic marks as any other such concatenation. Thus a source of possible ambiguity has been avoided.

Although the recursive nature of some of the metaproduction rules makes it possible to produce terminal metaproductions of arbitrary length,

the length of the terminal metaproductions necessarily involved in the production of any given **program** is finite.}

1.1.3.4. Hyper-rules and consistent substitution

a) The hyper-rules {b} of the language are those given in the sections of this Report whose heading begins with "Syntax".

b) A "hyper-rule" consists of the following items, in order:
an optional asterisk ;
a nonempty hypernotion H ;
a colon ;
a nonempty sequence of "hyperalternatives" separated by semicolons ;
a point.
It is said to be a hyper-rule "for" {the hypernotion (1.1.3.1.e)} H.

c) A "hyperalternative" is a nonempty sequence of hypernotions separated by commas.

{Examples:

b) **NOTION sequence :**
**NOTION ; NOTION, NOTION sequence.** (1.3.3.b)
c) **NOTION, NOTION sequence** }

d) A production rule PR {1.1.3.2.b} is derived from a hyper-rule HR if a copy of HR can be transformed into a copy of PR by replacing the set of all the hypernotions in the copy by a "consistent substitute" {e} for that set.

e) A set of {one or more} protonotions PP is a "consistent substitute" for a corresponding set of hypernotions HH if a copy of HH can be transformed into a copy of PP by means of the following step:
Step: If the copy contains one or more metanotions then, for some terminal metaproduction T of one such metanotion M, each occurrence of M in the copy is replaced by a copy of T and the Step is taken again.
{See 1.1.4.1.a for another application of consistent substitution.}

{Applying this derivation process to the hyper-rule given above (c) may give rise to

**digit cypher sequence :**
**digit cypher ; digit cypher, digit cypher sequence.**

which is therefore a production rule of the language. Note that

**digit cypher sequence :**
**digit cypher ; digit cypher, letter b sequence.**

is not a production rule of the language, since the replacement of the metanotion "**NOTION**" by one of its terminal metaproductions must be consistent throughout.}

{Since some metanotions have an infinite number of terminal metaproductions, the number of production rules which may be derived is infinite. The language is, however, so designed that, for the production of any **program** of finite length, only a finite number of those production rules is needed.}

{f) The rules under Syntax are provided with "cross-references" to be interpreted as follows.

Each hypernotion H of a hyperalternative of a hyper-rule A is followed by a reference to those hyper-rules B whose derived production rules are for notions which could be substituted for that H. Likewise, the hypernotions of each hyper-rule B are followed by a reference back to A. Alternatively, if H is to be replaced by a symbol, then it is followed by a reference to its representation in section 9.4.1. Moreover, in some cases, it is more convenient to give a cross-reference to one metaproduction rule rather than to many hyper-rules, and in these cases the missing cross-references will be found in the metaproduction rule.

Such a reference is, in principle, the section number followed by a letter indicating the line where the rule or representation appears, with the following conventions:

(i) the references whose section number is that of the section in which they appear are given first and their section number is omitted; e.g., "8.2.1.a" appears as "a" in section "8.2.1";
(ii) all points and a final 1 are omitted, and 10 appears as A; e.g., "8.2.1.a" appears as "82a" elsewhere and "10.3.4.1.1.i" appears as "A341i";
(iii) a section number which is the same as that of the preceding reference is omitted; e.g., "82a,82b,82c" appears as "82a,b,c";
(iv) the presence of a blind alley derived from that hypernotion is indicated by "-"; e.g., in 8.0.1.a after "**MOID denotation**", since "**MOID**" may be replaced by, for example, '**reference to real**', but '**reference to real denotation**' is not a notion.}

### 1.1.4. The semantics

The "meaning" of **programs** {2.2.1.a} in the strict language is defined in the semantics by means of sentences {in somewhat formalized natural language} which specify the "actions" to be carried out during the "elaboration" {2.1.4.1} of those **programs**. The "meaning" of a **program** in a representation language is the same as the meaning of the **program** in the strict language which it represents {9.3}.

{The semantics makes extensive use of hypernotions and paranotions in order to "designate", respectively, protonotions and constructs. The word "designate" should be understood in the sense that the word "flamingo" may "designate" any animal of the family *Phoenicopteridae*.}

1.1.4.1. Hypernotions, designation and envelopment

{Hypernotions, when enclosed between apostrophes, are used to "designate" protonotions belonging to certain classes; e.g., **'LEAP'** designates any of the protonotions **'local'**, **'primal'** and **'heap'**.}

a) Hypernotions standing in the text of this Report, except those in hyper-rules {1.1.3.4.b} or metaproduction rules {1.1.3.3.b}, "designate" any protonotions which may be consistently substituted {1.1.3.4.e} for them, the consistent substitution being applied over all the hypernotions contained in each complete sub-section of the text (a sub-section being one of the lettered sub-divisions, if any, or else the whole, of a numbered section).

{Thus **'QUALITY TAX'** is a hypernotion designating protonotions such as **'integral letter i'**, **'real letter x'**, etc. If, in some particular discussion, it in fact designates **'integral letter i'**, then all occurrences of "**QUALITY**" in that subsection must, over the span of that discussion, designate **'integral'** and all occurrences of "**TAX**" must designate **'letter i'**. It may then be deduced from subsection 4.8.2.a that in order, for example, to "ascribe to an **integral-defining-indicator-with-letter-i**", it is **'integral letter i'** that must be "made to access V inside the locale".}

Occasionally, where the context clearly so demands, consistent substitution may be applied over less than a section. {For example, in the introduction to section 2.1.1.2, there are several occurrences of "'**MOID**'", of which two are stated to designate specific (and different) protonotions spelled out in full, and of which others occur in the plural form "'**MOID**'s", which is clearly intended to designate a set of different members of the class of terminal metaproductions of "**MOID**".}

b) If a protonotion (a hypernotion) P consists of the concatenation of the protonotions (hypernotions) A, B and C, where A and C are possibly empty, then P "contains" B at the position {in P} determined by the length of A. {Thus, **'abcdefcdgh'** contains **'cd'** at its third and seventh positions.}

c) A protonotion P1 "envelops" a protonotion P2 as specifically designated by a hypernotion H2 if P2, or some equivalent {2.1.1.2.a} of it, is contained {b} at some position within P1 but not, at that position, within any different {intermediate} protonotion P3 also contained in P1 such that H2 could also designate P3.

{Thus the **'MODE'** enveloped by **'reference to real closed clause'** is **'reference to real'** rather than **'real'**; moreover, the mode (2.1.1.2.b) specified by ***struct (real a, struct (bool b, char c) d)*** envelops **'FIELD'** just twice.}

1.1.4.2. Paranotions

{In order to facilitate discussion, in this Report, of constructs with specified originals, the concept of a "paranotion" is introduced. A paranotion is a noun that designates constructs (1.1.3.2.e); its meaning is

not necessarily that found in a dictionary but can be construed from the rules which follow.)

a) A "paranotion" P is a hypernotion (not between apostrophes) which is used, in the text of this Report, to "designate" any construct whose original O satisfies the following:

- P, regarded as a hypernotion (i.e., as if it had been enclosed in apostrophes), designates (1.1.4.1.a) an "abstraction" (b) of O.

(For example, the paranotion "**fixed-point-numeral**" could designate the construct represented by *02*, since, had it been in apostrophes, it would have designated an abstraction of the notion '**fixed point numeral**', which is the original of that construct. However, that same representation could also be described as a **digit-cypher-sequence**, and as such it would be a direct descendent of that **fixed-point-numeral**.)

(As an aid to the reader in distinguishing them from other hypernotions, paranotions are not enclosed between apostrophes and are provided with hyphens where, otherwise, they would have been provided with blanks.)

The meaning of a paranotion to which the small syntactic mark "**s**" has been appended is the same as if the letter "s" (which is in the same type font as the marks in this sentence) had been appended instead. (Thus the **fixed-point-numeral** *02* may be said to contain two **digit-cyphers**, rather than two **digit-cyphers**.) Moreover, the "**s**" may be inserted elsewhere than at the end if no ambiguity arises (e.g., "**sources-for-MODINE**" means the same as "**source-for-MODINEs**").

An initial small syntactic mark of a paranotion is often replaced by the corresponding large syntactic mark (in order to improve readability, as at the start of a sentence) without change of meaning (; e.g., "**Identifier**" means the same as "**identifier**").

b) A protonotion P2 is an "abstraction" of a protonotion P1 if

(i) P2 is an abstraction of a notion whose production rule begins with an asterisk and of which P1 is an alternative

(e.g., '**trimscript**' (5.3.2.1.h) is an abstraction of any of the notions designated by '**NEST trimmer**', '**NEST subscript**' and '**NEST revised lower bound option**'), or

(ii) P1 envelops a protonotion P3 which is designated by one of the "elidible hypernotions" listed in section c below, and P2 is an abstraction of the protonotion consisting of P1 without that enveloped P3

(e.g., '**choice using boolean start**' is an abstraction of the notions '**choice using boolean brief start**' and '**choice using boolean bold start**' (by elision of a '**STYLE**' from 9.1.1.a)), or

(iii) P2 is equivalent to {2.1.1.2.a} P1
{e.g., **'bold begin symbol'** is an abstraction of **'bold begin symbol'**}.

{For an example invoking all three rules, it may be observed that **'union of real integral mode defining indicator'** is an abstraction of some **'union of integral real mode NEST defining identifier with letter a'** (4.8.1.a). Note, however, that **'choice using union of integral real mode brief start'** is not an abstraction of the notion **'choice using union of integral real boolean mode brief start'**, because the **'boolean'** that has apparently been elided is not an enveloped **'MOID'** of that notion.}

c) The "elidible hypernotions" mentioned in section b above are the following:

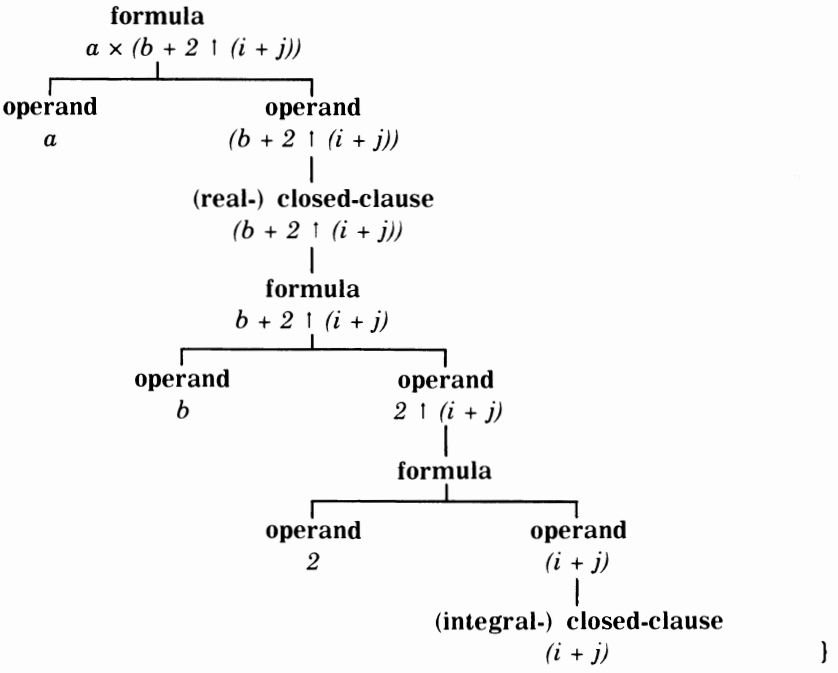
**"STYLE" • "TALLY" • "LEAP" • "DEFIED" • "VICTAL" • "SORT" • "MOID" • "NEST" • "REFETY routine" • "label" • "with TAX" • "with DECSETY LABSETY" • "of DECSETY LABSETY" • "defining LAYER"**.

{Which one of several possible notions or symbols is the original of a construct designated by a given paranotion will be apparent from the context in which that paranotion appears. For example, when speaking of the **formal-declarer** of an **identity-declaration**, if the **identity-declaration** is one whose terminal production (1.1.3.2.f) happens to be ***ref real*** *x* = ***loc real***, then the original of that **formal-declarer** is some notion designated by **'formal reference to real NEST declarer'**.}

{Since a paranotion designates a construct, all technical terms which are defined for constructs can be used with paranotions without formality.}

d) If two paranotions P and Q designate, respectively, two constructs S and T, and if S is a descendent of T, then P is termed a "constituent" of Q unless there exists some {intermediate construct} U such that
(i) S is a descendent of U,
(ii) U is a descendent of T, and
(iii) either P or Q could {equally well} designate U.

{Hence *a* (S1) is a constituent **operand** of the **formula** $a \times (b + 2 \uparrow (i + j))$ (T), but *b* (S2) is not, since it is a descendent of an intermediate **formula** $b + 2 \uparrow (i + j)$ (U), which is itself descended from T. Likewise, $(b + 2 \uparrow (i + j))$ is a constituent **closed-clause** of the **formula** T, but the **closed-clause** $(i + j)$ is not, because it is descended from an intermediate **closed-clause**. However, $(i + j)$ is a constituent **integral-closed-clause** of T, because the intermediate **closed-clause** is, in fact, a **real-closed-clause**.




}

1.1.4.3. Undefined

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

{A distinction must be drawn between the yielding of an undefined value (whereupon elaboration continues with possibly unpredictable results) and the complete undefinedness of the further elaboration. The action to be taken in this latter case is at the discretion of the implementer, and may be some form of continuation (but not necessarily the same as any other implementer's continuation), or some form of interruption (2.1.4.3.h) brought about by some run-time check.}

b) If some condition is "required" to be satisfied during some elaboration then, if it is not so satisfied, the further elaboration is undefined.

c) A "meaningful" **program** is a **program** {2.2.1.a} whose elaboration is defined by this Report.

{Whether all **programs**, only **particular-programs**, only meaningful **programs**, or even only meaningful **particular-programs** are "ALGOL 68" **programs** is a matter for individual taste.}

1.1.5. Translations and variants

a) The definitive version D of this Report is written in English. A translation T of this Report into some other language is an acceptable translation if:

- T defines the same set of production trees as D, except that
  (i) the originals contained in each production tree of T may be different protonotions obtained by some uniform translation of the corresponding originals contained in the corresponding production tree of D, and
  (ii) descendents of those production trees need not be the same if their originals are predicates {1.3.2};

- T defines the meaning {2.1.4.1.a} of each of its **programs** to be the same as that of the corresponding **program** defined by D;

- T defines the same reference language {9.4} and the same standard environment {10} as D;

- T preserves, under another mode of expression, the meaning of each section of D except that:
  (i) different syntactic marks {1.1.3.1.a} may be used {with a correspondingly different metaproduction rule for "**ALPHA**" (1.3.1.B)};
  (ii) the method of derivation of the production rules {1.1.3.4} and their interpretation {1.1.3.2} may be changed to suit the peculiarities of the particular natural language {; e.g., in a highly inflected natural language, it may be necessary to introduce some inflections into the hypernotions, for which changes such as the following might be required:
    1) additional means for the creation of extra metaproduction rules (1.1.3.3.a);
    2) a more elaborate definition of "consistent substitute" (1.1.3.4.e);
    3) a more elaborate definition of "equivalence" between protonotions (2.1.1.2.a);
    4) different inflections for paranotions (1.1.4.2.a)};
  (iii) some pragmatic remarks {1.1.2} may be changed.

b) A version of this Report may, additionally, define a "variant of ALGOL 68" by providing:
  (i) additional or alternative representations in the reference language {9.4},
  (ii) additional or alternative rules for the notion '**character glyph**' {8.1.4.1.c} and for the metanotions "**ABC**" {9.4.2.1.L} and "**STOP**" {10.1.1.B},
  (iii) additional or alternative declarations in the standard environment which must, however, have the same meaning as the ones provided in D;
provided always that such additional or alternative items are delineated in the text in such a way that the original language, as defined in D, is still defined therein.

## 1.2. General metaproduction rules

{The reader may find it helpful to note that a metanotion ending in "**ETY**" always has "**EMPTY**" as one of the hypernotions on its right-hand side.}

1.2.1. Metaproduction rules of modes

A) **MODE :: PLAIN ; STOWED ; REF to MODE ; PROCEDURE ; UNITED ; MU definition of MODE ; MU application.**
B) **PLAIN :: INTREAL ; boolean ; character.**
C) **INTREAL :: SIZETY integral ; SIZETY real.**
D) **SIZETY :: long LONGSETY ; short SHORTSETY ; EMPTY.**
E) **LONGSETY :: long LONGSETY ; EMPTY.**
F) **SHORTSETY :: short SHORTSETY ; EMPTY.**
G) **EMPTY :: .**
H) **STOWED :: structured with FIELDS mode ; FLEXETY ROWS of MODE.**
I) **FIELDS :: FIELD ; FIELDS FIELD.**
J) **FIELD :: MODE field TAG{942A}.**
K) **FLEXETY :: flexible ; EMPTY.**
L) **ROWS :: row ; ROWS row.**
M) **REF :: reference ; transient reference.**
N) **PROCEDURE :: procedure PARAMETY yielding MOID.**
O) **PARAMETY :: with PARAMETERS ; EMPTY.**
P) **PARAMETERS :: PARAMETER ; PARAMETERS PARAMETER.**
Q) **PARAMETER :: MODE parameter.**
R) **MOID :: MODE ; void.**
S) **UNITED :: union of MOODS mode.**
T) **MOODS :: MOOD ; MOODS MOOD.**
U) **MOOD :: PLAIN ; STOWED ; reference to MODE ; PROCEDURE ; void.**
V) **MU :: muTALLY.**
W) **TALLY :: i ; TALLY i.**

{The metaproduction rule for "TAG" is given in section 9.4.2.1. It suffices for the present that it produces an arbitrarily large number of terminal metaproductions.}

1.2.2. Metaproduction rules associated with phrases and coercion

A) **ENCLOSED :: closed ; collateral ; parallel ; CHOICE{34A} ; loop.**
B) **SOME :: SORT MOID NEST.**
C) **SORT :: strong ; firm ; meek ; weak ; soft.**

1.2.3. Metaproduction rules associated with nests

A) **NEST :: LAYER ; NEST LAYER.**
B) **LAYER :: new DECSETY LABSETY.**
C) **DECSETY :: DECS ; EMPTY.**
D) **DECS :: DEC ; DECS DEC.**
E) **DEC :: MODE TAG{942A} ; priority PRIO TAD{942F} ; MOID TALLY TAB{942D} ; DUO TAD{942F} ; MONO TAM{942K}.**
F) **PRIO :: i ; ii ; iii ; iii i ; iii ii ; iii iii ; iii iii i ; iii iii ii ; iii iii iii.**
G) **MONO :: procedure with PARAMETER yielding MOID.**

H) **DUO :: procedure with PARAMETER1 PARAMETER2 yielding MOID.**
I) **LABSETY :: LABS ; EMPTY.**
J) **LABS :: LAB ; LABS LAB.**
K) **LAB :: label TAG**{942A}.

{The metaproduction rules for "**TAB**", "**TAD**" and "**TAM**" are given in section 9.4.2.1. It suffices for the present that each of them produces an arbitrarily large number of terminal metaproductions, none of which is a terminal metaproduction of "**TAG**".}

{"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. General hyper-rules

{Predicates are used in the syntax to enforce certain restrictions on the production trees, such as that each **applied-indicator** should identify a uniquely determined **defining-indicator**. A more modest use is to reduce the number of hyper-rules by grouping several similar cases as alternatives in one rule. In these cases predicates are used to test which alternative applies.}

### 1.3.1. Syntax of general predicates

A) **NOTION :: ALPHA ; NOTION ALPHA.**
B) **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.**
C) **NOTETY :: NOTION ; EMPTY.**
D) **THING :: NOTION ; (NOTETY1) NOTETY2 ; THING (NOTETY1) NOTETY2.**
E) **WHETHER :: where ; unless.**

a) **where true : EMPTY.**
b) **unless false : EMPTY.**
c) **where THING1 and THING2 : where THING1, where THING2.**
d) **where THING1 or THING2 : where THING1 ; where THING2.**
e) **unless THING1 and THING2 : unless THING1 ; unless THING2.**
f) **unless THING1 or THING2 : unless THING1, unless THING2.**

g) **WHETHER (NOTETY1) is (NOTETY2) : WHETHER (NOTETY1) begins with (NOTETY2)**{h,i,j} **and (NOTETY2) begins with (NOTETY1)**{h,i,j}.
h) **WHETHER (EMPTY) begins with (NOTION)**{g,j} : **WHETHER false**{b,-}.
i) **WHETHER (NOTETY) begins with (EMPTY)**{g,j} : **WHETHER true**{a,-}.

j) **WHETHER (ALPHA1 NOTETY1) begins with**
**(ALPHA2 NOTETY2)**{g,j,m} :
**WHETHER (ALPHA1) coincides with (ALPHA2) in**
**(abcdefghijklmnopqrstuvwxyz)**{k,l,-}
**and (NOTETY1) begins with (NOTETY2)**{h,i,j}.

k) **where (ALPHA) coincides with (ALPHA) in (NOTION)**{j} :
**where true**{a}.

l) **unless (ALPHA1) coincides with (ALPHA2) in (NOTION)**{j} :
**where (NOTION) contains (ALPHA1 NOTETY ALPHA2)**{m}
**or (NOTION) contains (ALPHA2 NOTETY ALPHA1)**{m}.

m) **WHETHER (ALPHA NOTETY) contains (NOTION)**{l,m} :
**WHETHER (ALPHA NOTETY) begins with (NOTION)**{j}
**or (NOTETY) contains (NOTION)**{m,n}.

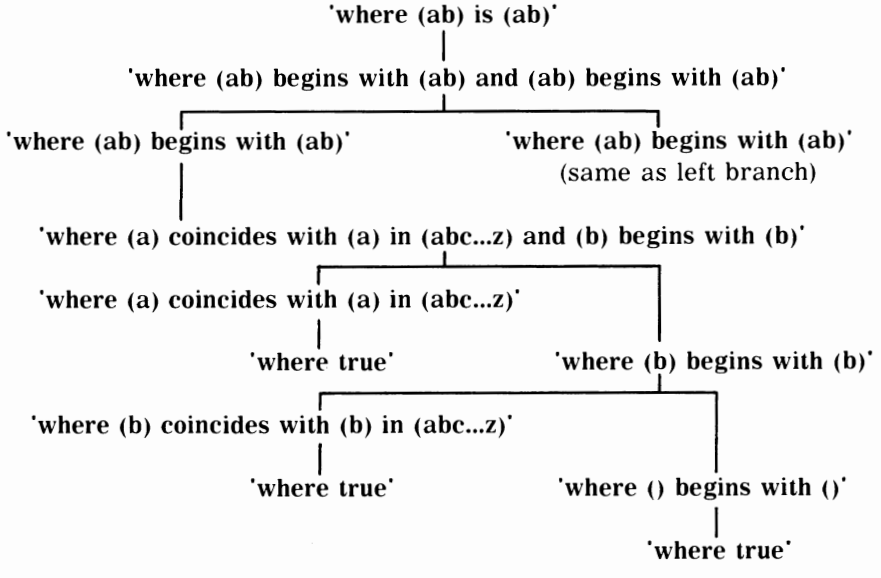
n) **WHETHER (EMPTY) contains (NOTION)**{m} : **WHETHER false**{b,-}.

{The small syntactic marks "(" and ")" are used to ensure, in a simple way, the unambiguous application of these predicates.}

1.3.2. The holding of predicates

A "predicate" is a protonotion which begins with **'where'** or **'unless'** {unified into **'WHETHER'**}. For a predicate P, either one or more production trees may be produced {1.1.3.2.f} {all of which are then invisible}, in which case P "holds", or no production tree may be produced {since each attempt to produce one runs into blind alleys}, and then P "does not hold".

{For example, the predicate **'where (ab) is (ab)'** holds. Its production tree may be depicted thus:



If a predicate holds, then its production tree always terminates via **'where true'** or **'unless false'**. If it does not hold, then, in general, the blind alleys are **'where false'** and **'unless true'**. Although almost all the hyper-rules concerned are for hypernotions beginning with "**WHETHER**" and so provide, each time, production rules for pairs of predicates such as **'where THING1'** and **'unless THING1'**, this does not mean that in each such case one of the pair must hold. For example, **'where digit four counts iii'** (4.3.1.c) does not hold, but no care has been taken to make **'unless digit four counts iii'** hold either, since there is no application for it in this Report.

In the semantics, no meaning is ascribed to constructs whose originals are predicates. They serve purely syntactical purposes.}

1.3.3. Syntax of general constructions

A) **STYLE :: brief ; bold ; style TALLY..**

a) **NOTION option : NOTION ; EMPTY.**
b) **NOTION sequence{b) : NOTION ; NOTION, NOTION sequence{b).**
c) **NOTION list{c) :**
**NOTION ; NOTION, and also{94f) token, NOTION list{c).**
d) **NOTETY STYLE pack :**
**STYLE begin{94f,-) token, NOTETY, STYLE end{94f,-) token.**
e) **NOTION STYLE bracket :**
**STYLE sub{94f,-) token, NOTION, STYLE bus{94f,-) token.**
f) **THING1 or alternatively THING2 : THING1 ; THING2.**

{It follows from this syntax that production rules such as
**digit cypher sequence :**
**digit cypher ; digit cypher, digit cypher sequence.**
(which was used in the production of the example in 1.1.3.2.f, but for which no more explicit hyper-rule is given) are immediately available. Thus the number of hyper-rules actually written in this Report has been reduced and those that remain have, hopefully, been made more readable, since these general constructions are so worded as to suggest what their productions should be.

For this reason, cross-references (1.1.3.4.f) to these rules have been replaced by more helpful references; e.g., in 8.1.1.1.b, instead of "**digit cypher sequence{133b)**", the more helpful "**digit cypher{c) sequence**" is given. Likewise, references within the general constructions themselves have been restricted to a bare minimum.}

## 2. The computer and the program

The meaning of a **program** in the strict language is explained in terms of a hypothetical computer which performs the set of actions {2.1.4} which constitute the elaboration {2.1.4.1} of that **program**. The computer deals with a set of "objects" {2.1.1}.

## 2.1. 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.}

### 2.1.1. Objects

An "object" is a construct {1.1.3.2.e}, a "value" {2.1.1.1.a}, a "locale" {2.1.1.1.b}, an "environ" {2.1.1.1.c} or a "scene" {2.1.1.1.d}.

{Constructs may be classified as "external objects", since they correspond to the text of the **program**, which, in a more realistic computer, would be compiled into some internal form in which it could operate upon the "internal objects", namely the values, the locales, the environs and the scenes. However, the hypothetical computer has no need of a compilation phase, it being presumed able to examine the **program** and all of its descendent constructs at the same time as it is manipulating the internal objects.}

#### 2.1.1.1. Values, locales, environs and scenes

a) A "value" is a "plain value" {2.1.3.1}, a "name" {2.1.3.2}, a "stowed value" (i.e., a "structured value" {2.1.3.3} or a "multiple value" {2.1.3.4}) or a "routine" {2.1.3.5}.

{For example, a real number is a plain value. A special font is used for values appearing in the text of this Report, thus: 3.14, true. This is not to be confused with the italic and bold fonts used for constructs. This same special font is also used for letters designating such things as constructs and protonotions.}

b) A "locale" {is an internal object which} corresponds to some **'DECSETY LABSETY'** {1.2.3.C,I}. A "vacant locale" is one for which that **'DECSETY LABSETY'** is **'EMPTY'**.

{Each **'QUALITY TAX'** (4.8.1.F,G) enveloped by that **'DECSETY LABSETY'** corresponds to a **QUALITY-defining-indicator-with-TAX** (i.e., to an **identifier**, **operator** or **mode-indication**) declared in the construct whose elaboration caused that locale to be created. Such a **'QUALITY TAX'** may be made to "access" a value or a scene "inside" that locale (2.1.2.c).

A locale may be thought of as a number of storage cells, into which such accessed objects are placed.}

{The terminal metaproductions of the metanotions "**DEC**", "**LAB**" and "**FIELD**" (or of the more frequently used "**PROP**", which includes them all) are all of the form **'QUALITY TAX'**. These "properties" are used in the syntax and semantics concerned with nests and locales in order to associate, in a particular situation, some quality with that **'TAX'**.}

c) An "environ" is either empty, or is composed of an environ and a locale.

{Hence, each environ is derived from a series of other environs, stemming ultimately from the empty "primal environ" in which the **program** is elaborated (2.2.2.a).}

d) A "scene" S is an object which is composed of a construct C {1.1.3.2.e} and an environ E. C is said to be the construct, and E the environ, "of" S.

{Scenes may be accessed inside locales (2.1.2.c) by **'LAB**'s or **'DEC**'s arising from **label-identifiers** or from **mode-indications**, and they may also be values (2.1.3.5).}

2.1.1.2. Modes

{Each value has an attribute, termed its "mode", which defines how that value relates to other values and which actions may be applied to it. This attribute is described, or "spelled", by means of some **'MOID'** (1.2.1.R) (thus there is a mode spelled **'real'**, and there is a mode spelled **'structured with real field letter r letter e real field letter i letter m mode'**). Since it is intended that the modes specified by the **mode-indications** ***a*** and ***b*** in

***mode a*** = ***struct*** (***ref a*** *b*),
***mode b*** = ***struct*** (***ref struct*** (***ref b*** *b*) *b*)

should in fact be the same mode, it is necessary that both the **'MOID'**

**'mui definition of structured with reference to mui application field letter b mode'**

and the **'MOID'**

**'muii definition of structured with reference to structured with reference to muii application field letter b mode field letter b mode'**

(and indeed many others) should be alternative spellings of that same mode. Similarly, the mode specified by the **declarer** ***union*** (***int***, ***real***) may be spelled as either **'union of integral real mode'** or **'union of real integral mode'**. All those **'MOID**'s which are spellings of one same mode are said to be "equivalent to" one another (a).

Certain **'MOID**'s, such as **'reference to muiii application'**, **'reference to muiiii definition of reference to muiiii application'**, **'union of real reference to real mode'**, and **'structured with integral field letter a real field letter a mode'**, are ill formed (7.4, 4.7.1.f, 4.8.1.c) and do not spell any mode.

Although for most practical purposes a "mode" can be regarded as simply a **'MOID'**, its rigorous definition therefore involves the whole class of **'MOID**'s, equivalent to each other, any of which could describe it.}

a) **'MOID1'** {1.2.1.R} is "equivalent to" **'MOID2'** if the predicate **'where MOID1 equivalent MOID2'** {7.3.1.a} holds {1.3.2}.

{A well formed **'MOID'** is always equivalent to itself; **'union of integral real mode'** is equivalent to **'union of real integral mode'**.}

A protonotion P is "equivalent to" a protonotion Q if it is possible to transform a copy Pc of P into a copy Qc of Q in the following step:
Step: If Pc is not identical to Qc, then some '**MOID1**' contained in Pc, but not within any (larger) '**MOID2**' contained in Pc, is replaced by some equivalent '**MOID**', and the Step is taken again.
(Thus '**union of integral real mode identifier**' is equivalent to '**union of real integral mode identifier**'.)

b) A "mode" is a class C of '**MOID**'s such that each member of C is equivalent (a) to each other member of C and also to itself (in order to ensure well formedness), but not to any '**MOID1**' which is not a member of C.
(However, it is possible (except when equivalence of modes is specifically under discussion) to discuss a mode as if it were simply a terminal metaproduction of "**MOID**", by virtue of the abbreviation to be given in 2.1.5.f.)

c) Each value is of one specific mode.
(For example, the mode of the value 3.14 is '**real**'. However, there are no values whose mode begins with '**union of**', '**transient reference to**' or '**flexible ROWS of**' (see 2.1.3.6).)

2.1.1.3. Scopes

(A value V may "refer to" (2.1.2.e), or be composed from (2.1.1.**1.d**), another internal object O (e.g., a name may refer to a value; a routine, which is a scene, is composed, in part, from an environ). Now the lifetime of the storage cells containing (2.1.3.2.a) or implied by (2.1.1.1.b) O may be limited (in order that they may be recovered after a certain time), and therefore it must not be possible to preserve V beyond that lifetime, for otherwise an attempt to reach some no-longer-existent storage cell via V might still be made. This restriction is expressed by saying that, if V is to be "assigned" (5.2.1.2.b) to some name W, then the "scope" of W must not be "older" than the scope of V. Thus, the scope of V is a measure of the age of those storage cells, and hence of their lifetime.)

a) Each value has one specific "scope" (which depends upon its mode or upon the manner of its creation; the scope of a value is defined to be the same as that of some environ).

b) Each environ has one specific "scope". (The scope of each environ is "newer" (2.1.2.f) than that of the environ from which it is composed (2.1.1.1.c).)
(The scope of an environ is not to be confused with the scopes of the values accessed inside its locale. Rather, the scope of an environ is used when defining the scope of scenes for which it is necessary (7.2.2.c) or of the yields of generators for which it is "local" (5.2.3.2.b). The scope of an environ is defined relative (2.1.2.f) to the scope of some other environ, so that hierarchies of scopes are created depending ultimately upon the scope of the primal environ (2.2.2.a).)

2.1.2. Relationships

a) Relationships either are "permanent", i.e., independent of the **program** and of its elaboration, or actions may cause them to "hold" or to cease to hold. Relationships may also be "transitive"; i.e., if "*" is such a relationship and A*B and B*C hold, then A*C holds also.

b) "To be the yield of" is a relationship between a value and an action, viz., the elaboration of a scene. This relationship is made to hold upon the completion of that elaboration {2.1.4.1.b}.

c) "To access" is a relationship between a **'PROP'** {4.8.1.E} and a value or a scene V which may hold "inside" some specified locale L (whose **'DECSETY LABSETY'** envelops **'PROP'**). This relationship is made to hold when **'PROP'** is "made to access" V inside L {3.5.2.Step 4, 4.8.2.a} and it then holds also between any **'PROP1'** equivalent to {2.1.1.2.a} **'PROP'** and V inside L.

d) The permanent relationships between values are: "to be of the same mode as" {2.1.1.2.c}, "to be smaller than", "to be widenable to", "to be lengthenable to" {2.1.3.1.e} and "to be equivalent to" {2.1.3.1.g}. If one of these relationships is defined at all for a given pair of values, then it either holds or does not hold permanently. These relationships are all transitive.

e) "To refer to" is a relationship between a "name" {2.1.3.2.a} N and some other value. This relationship is made to hold when N is "made to refer to" that value and ceases to hold when N is made to refer to some other value.

f) There are three transitive relationships between scopes, viz., a scope A {2.1.1.3} may be either "newer than", or "the same as" or "older than" a scope B. If A is newer than B, then B is older than A and vice-versa. If A is the same as B, then A is neither newer nor older than B (but the converse is not necessarily true, since the relationship is not defined at all for some pairs of scopes).

g) "To be a subname of" is a relationship between a name and a "stowed name" {2.1.3.2.b}. This relationship is made to hold when that stowed name is "endowed with subnames" {2.1.3.3.e, 2.1.3.4.g} or when it is "generated" {2.1.3.4.j,l}, and it continues to hold until that stowed name is endowed with a different set of subnames.

2.1.3. Values

2.1.3.1. Plain values

a) A plain value is either an "arithmetic value", i.e., an "integer" or a "real number", or is a "truth value" {f}, a "character" {g} or a "void value" {h}.

b) An arithmetic value has a "size", i.e., an integer characterizing the degree of discrimination with which it is kept in the computer.

c) The mode of an integer or of a real number of size n is, respectively, some **'SIZETY integral'** or **'SIZETY real'** where, if n is positive (zero, negative), that **'SIZETY'** is n times **'long'** (is empty, is -n times **'short'**).

d) The number of integers or of real numbers of a given size that can be distinguished increases (decreases) with that size until a certain size is reached, viz., the "number of extra lengths" (minus the "number of extra shorths") of integers or of real numbers, respectively, {10.2.1.a,b,d,e} after which it is constant.

{Taking Three as the subject to reason about-
A convenient number to state- }

e) For the purpose of explaining the meaning of the widening coercion and of the **operators** declared in the **standard-prelude**, the following properties of arithmetic values are assumed:

- for each pair of integers or of real numbers of the same size, the relationship "to be smaller than" is defined with its usual mathematical meaning {10.2.3.3.a, 10.2.3.4.a};
- for each pair of integers of the same size, a third distinguishable integer of that size may exist, the first integer "minus" the other {10.2.3.3.g};

{We add Seven, and Ten, and then multiply out
By One Thousand diminished by Eight. }

- for each pair of real numbers of the same size, three distinguishable real numbers of that size may exist, the first real number "minus" ("times", "divided by") the other one {10.2.3.4.g,l,m};
- in the foregoing, the terms "minus", "times" and "divided by" have their usual mathematical meaning but, in the case of real numbers, their results are obtained "in the sense of numerical analysis", i.e., by performing those operations on numbers which may deviate slightly from the given ones {; this deviation is left undefined in this Report};

{The result we proceed to divide, as you see,
By Nine Hundred and Ninety and Two }

- each integer of a given size is "widenable to" a real number close to it and of that same size {6.5};
- each integer (real number) of a given size can be "lengthened to" an integer (real number) close to it whose size is greater by one {10.2.3.3.q, 10.2.3.4.n}.

f) A "truth value" is either "true" or "false". Its mode is '**boolean**'.

{Then subtract Seventeen, and the answer must be
Exactly and perfectly true.
The Hunting of the Snark, Lewis Carroll.}

g) Each "character" is "equivalent" to a nonnegative integer of size zero, its "integral equivalent" {10.2.1.n}; this relationship is defined only to the extent that different characters have different integral equivalents, and that there exists a "largest integral equivalent" {10.2.1.p}. The mode of a character is '**character**'.

h) The only "void value" is "empty". Its mode is '**void**'.

{The elaboration of a construct yields a void value when no more useful result is needed. Since the syntax does not provide for **void-variables**, **void-identity-declarations** or **void-parameters**, the programmer cannot make use of void values, except those arising from uniting (6.4).}

i) The scope of a plain value is the scope of the primal environ {2.2.2.a}.

2.1.3.2. Names

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

a) A "name" is a value which can be "made to refer to" {d, 5.2.3.2.a, 5.2.1.2.b} some other value, or which can be "nil" {and then refers to no value}; moreover, for each mode beginning with '**reference to**', there is exactly one nil name of that mode.

A name may be "newly created" {by the elaboration of a **generator** (5.2.3.2) or a **rowed-to-FORM** (6.6.2), when a stowed name is endowed with subnames (2.1.3.3.e, 2.1.3.4.g) and, possibly, when a name is "generated" (2.1.3.4.j, l)}. The name so created is different from all names already in existence.

{A name may be thought of as the address of the storage cell or cells, in the computer, used to contain the value referred to. The creation of a name implies the reservation of storage space to hold that value.}

b) The mode of a name N is some '**reference to MODE**' and any value which is referred to by N must be "acceptable to" {2.1.3.6.d} that '**MODE**'. If '**MODE**' is some '**STOWED**', then N is said to be a "stowed name".

c) The scope of a name is the scope of some specific environ {usually the "local environ" (5.2.3.2.b) of some **generator**}. The scope of a name which is nil is the scope of the primal environ {2.2.2.a}.

d) If N is a stowed name referring to a structured (multiple) value V {2.1.3.3, 2.1.3.4}, and if a subname {2.1.2.g} of N selected {2.1.3.3.e, 2.1.3.4.g} by a **'TAG'** (an index) I is made to refer to a {new} value X, then N is made to refer to a structured (multiple) value which is the same as V except for its field (element) selected by I, which is {now made to be} X.

{For the mode of a subname, see 2.1.3.3.d and 2.1.3.4.f.}

2.1.3.3. Structured values

a) A "structured value" is composed of a sequence of other values, its "fields", each of which is "selected" {b} by a specific **'TAG'** {9.4.2.1.A}. {For the selection of a field by a **field-selector**, see 2.1.5.g.}

{The ordering of the fields of a structured value is utilized in the semantics of **structure-displays** (3.3.2.b) and **format-texts** (10.3.4), and in straightening (10.3.2.3.c).}

b) The mode of a structured value V is some **'structured with FIELDS mode'**. If the n-th **'FIELD'** enveloped by that **'FIELDS'** is some **'MODE field TAG'**, then the n-th field of V is "selected" by **'TAG'** and is acceptable to {2.1.3.6.d} **'MODE'**.

c) The scope of a structured value is the newest of the scopes of its fields.

d) If the mode of a name N {referring to a structured value} is some **'reference to structured with FIELDS mode'**, and if the predicate **'where MODE field TAG resides in FIELDS'** holds {7.2.1.b,c}, then the mode of the subname of N selected {e} by **'TAG'** is **'reference to MODE'**.

e) When a name N which refers to a structured value V is "endowed with subnames" {e, 2.1.3.4.g, 4.4.2.b, 5.2.3.2.a}, then,
For each **'TAG'** selecting a field F in V,

- a new subname M is created of the same scope as N;
- M is made to refer to F;
- M is said to be the name "selected" by **'TAG'** in N;
- if M is a stowed name {2.1.3.2.b}, then it is itself endowed with subnames {e, 2.1.3.4.g}.

2.1.3.4. Multiple values

a) A "multiple value" {of n dimensions} is composed of a "descriptor" and a sequence of other values, its "elements", each of which may be "selected" by a specific n-tuple of integers, its "index".

b) The "descriptor" is of the form
$((l_1, u_1), (l_2, u_2), \ldots, (l_n, u_n))$
where each $(l_i, u_i)$, i = 1, ... , n, is a "bound pair" of integers in which $l_i$ is the i-th "lower bound" and $u_i$ is the i-th "upper bound".

c) If for any i, i = 1, ... , n, $u_i < l_i$, then the descriptor is said to be "flat" and there is one element, termed a "ghost element" {, and not selected by any index; see also 5.2.1.2.b}; otherwise, the number of elements is $(u_1 - l_1 + 1) \times (u_2 - l_2 + 1) \times \ldots \times (u_n - l_n + 1)$ and each is selected by a specific index $(r_1, \ldots, r_n)$ where $l_i \leq r_i \leq u_i$, i = 1, ... , n.

d) The mode of a multiple value V is some **'ROWS of MODE'**, where that **'ROWS'** is composed of as many times **'row'** as there are bound pairs in the descriptor of V and where each element of V is acceptable to {2.1.3.6.d} that **'MODE'**.

{For example, given [ ] ***union (int, real)*** *ruir = (1, 2.0)*, the mode of the yield of *ruir* is **'row of union of integral real mode'**, the mode of its first element is **'integral'** and that of its second element is **'real'**.}

e) The scope of a multiple value is the newest of the scopes of its elements, if its descriptor is not flat, and, otherwise, is the scope of the primal environ {2.2.2.a}.

f) A multiple value, of mode **'ROWS of MODE'**, may be referred to either by a "flexible" name of mode **'reference to flexible ROWS of MODE1'**, or by a "fixed" name of mode **'reference to ROWS of MODE1'** where {in either case} **'MODE1'** "deflexes" {2.1.3.6.b} to **'MODE'**.

{The difference implies a possible difference in the method whereby the value is stored in the computer. The flexible case must allow a multiple value with different bounds to be assigned (5.2.1.2.b) to that name, whereas the fixed case can rely on the fact that those bounds will remain fixed during the lifetime of that name. Note that the "flexibility" is a property of the name: the underlying multiple value is the same value in both cases.}

If the mode of a name N {referring to a multiple value} is some **'reference to FLEXETY ROWS of MODE'**, then the mode of each subname of N is **'reference to MODE'**.

g) When a name N which refers to a multiple value V is "endowed with subnames" {g, 2.1.3.3.e, 4.4.2.b, 5.2.1.2.b, 5.2.3.2.a}, then,
For each index selecting an element E of V,

- a new subname M is created of the same scope as N;
- M is made to refer to E;
- M is said to be the name "selected" by that index in N;
- if M is a stowed name {2.1.3.2.b}, then it is itself endowed with subnames {g, 2.1.3.3.e}.

{In addition to the selection of an element (a) or a name (g) by means of an index, it is also possible to select a value, or to generate a new name referring to such a value, by means of a trim (h,i,j) or a **'TAG'** (k,l). Both indexes and trims are used in the elaboration of **slices** (5.3.2.2).}

h) A "trim" is an n-tuple, each element of which is either an integer {corresponding to a **subscript**} or a triplet (l, u, d) {corresponding to a **trimmer** or a **revised-lower-bound-option**}, such that at least one of those elements is a triplet {if all the elements are integers, then the n-tuple is an index (a)}. Each element of such a triplet is either an integer or is "absent".

{A trim (or an index) is yielded by the elaboration of an **indexer** (5.3.2.2.b).}

i) The multiple value W {of m dimensions} "selected" by a trim T in a multiple value V {of n dimensions, $1 \le m \le n$} is determined as follows:

- Let T be composed of integers and triplets $T_i$, i = 1, ... , n, of which m are actually triplets; let the j-th triplet be $(l_j, u_j, d_j)$, j = 1, ... , m;
- W is composed of

(i) a descriptor $((l_1 - d_1, u_1 - d_1), (l_2 - d_2, u_2 - d_2), \ldots, (l_m - d_m, u_m - d_m))$;

(ii) elements of V, where the element, if any, selected in W by an index $(w_1, \ldots, w_m)$ $\{l_j - d_j \le w_j \le u_j - d_j\}$ is that selected in V by the index $(v_1, \ldots, v_n)$ determined as follows:

For i = 1, ... , n,

Case A: $T_i$ is an integer:

- $v_i = T_i$;

Case B: $T_i$ is the j-th triplet $(l_j, u_j, d_j)$ of T:

- $v_i = w_j + d_j$.

j) The name M "generated" by a trim T from a name N which refers to a multiple value V is a {fixed} name, of the same scope as N, {not necessarily newly created} which refers to the multiple value W selected {i} by T in V. Each subname of M, as selected by an index Iw, is one of the {already existing} subnames of N, as selected by an index Iv, where each Iv is determined from T and the corresponding Iw using the method given in the previous sub-section.

k) The multiple value W "selected" by a **'TAG'** in a multiple value V {each of whose elements is a structured value} is composed of

(i) the descriptor of V, and

(ii) the fields selected by **'TAG'** in the elements of V, where the element, if any, selected in W by an index I is the field selected by **'TAG'** in the element of V selected by I.

l) The name M "generated" by a **'TAG'** from a name N which refers to a multiple value V {each of whose elements is a structured value} is a {fixed} name, of the same scope as N, {not necessarily newly created}

which refers to the multiple value selected {k} by **'TAG'** in V. Each subname of M selected by an index I is the {already existing} name selected {2.1.3.3.e} by **'TAG'** in the subname of N selected {g} by I.

2.1.3.5. Routines

a) A "routine" is a scene {2.1.1.1.d} composed of a **routine-text** {5.4.1.1.a,b} together with an environ {2.1.1.1.c}.

{A routine may be "called" (5.4.3.2.b), whereupon the **unit** of its **routine-text** is elaborated.}

b) The mode of a routine composed of a **PROCEDURE-routine-text** is **'PROCEDURE'**.

c) The scope of a routine is the scope of its environ.

2.1.3.6. Acceptability of values

a) {There are no values whose mode begins with **'union of'**. There exist names whose modes begin with **'reference to union of'**, e.g., *u* in ***union (int, real)*** *u;* . Here, however, *u*, whose mode is **'reference to union of integral real mode'**, refers either to a value whose mode is **'integral'** or to a value whose mode is **'real'**. It is possible to discover which of these situations obtains, at a given moment, by means of a **conformity-clause** (3.4.1.q).}

The mode **'MOID'** is "united from" the mode **'MOOD'** if **'MOID'** is some **'union of MOODSETY1 MOOD MOODSETY2 mode'**.

b) {There are no values whose mode begins with **'flexible'**. There exist flexible names whose modes begin with **'reference to flexible'**, e.g., *a1* in ***flex*** [*1* : *n*] ***real*** *a1;* . Here *a1*, whose mode is **'reference to flexible row of real'**, refers to a multiple value whose mode is **'row of real'** (see also 2.1.3.4.f). In general, there exist values only for those modes obtainable by "deflexing".}

The mode **'MOID1'** "deflexes" to the mode **'MOID2'** if the predicate **'where MOID1 deflexes to MOID2'** holds {4.7.1.a,b,c}.

{The deflexing process obtains **'MOID2'** by removing all **'flexible's** contained at positions in **'MOID1'** where they are not also contained in any **'REF to MOID3'**. Thus

**'structured with flexible row of character field letter a mode'**,

which is not the mode of any value, deflexes to

**'structured with row of character field letter a mode'**

which is therefore the mode of a value referable to by a flexible name of mode

**'reference to structured with flexible row of character field letter a mode'**.

This mode is already the mode of a name and therefore it cannot be deflexed any further.}

c) {There are no names whose mode begins with **'transient reference to'**.

The yield of a **transient-reference-to-MODE-FORM** is a "transient name" of mode **'reference to MODE'**, but, there being no **transient-reference-to-MODE-declarators** in the language (4.6.1), the syntax ensures that transient names can never be assigned, ascribed or yielded by the calling of a routine.

E.g., *xx* := *a1* [*i*] is not an **assignation** because *xx* is not a **reference-to-transient-reference-to-real-identifier**. Transient names originate from the slicing, multiple selection or rowing of a flexible name.}

d) A value of mode M1 is "acceptable to" a mode M2 if

(i) M1 is the same as M2, or

(ii) M2 is united {a} from M1 {thus the mode specified by ***union (real, int)*** accepts values whose mode is that specified by either ***real*** or ***int***}, or

(iii) M2 deflexes {b} to M1 {thus the mode **'flexible row of real'** (a mode of which there are no values) accepts values such as the yield of the **actual-declarer *flex*** [*1* : *n*] ***real*** which is a value of mode **'row of real'**}, or

(iv) M1 is some **'reference to MODE'** and M2 is **'transient reference to MODE'** {thus the mode **'transient reference to real'** accepts values (such as the yield of *a1* [*i*]) whose mode is **'reference to real'**}.

{See 2.1.4.1.b for the acceptability of the yield of a scene.}

## 2.1.4. Actions

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

### 2.1.4.1. Elaboration

a) The "elaboration" of certain scenes {those whose constructs are designated by certain paranotions} is specified in the sections of this Report headed "Semantics", which describe the sequence of "actions" which are to be carried out during the elaboration of each such scene.

{Examples of actions which may be specified are:

- the causing to hold of relationships,
- the creation of new names, and
- the elaboration of other scenes.}

The "meaning" of a scene is the effect of the actions carried out during its elaboration. Any of these actions or any combination thereof may be replaced by any action or combination which causes the same effect.

b) The elaboration of a scene S may "yield" a value. If the construct of S is a **MOID-NOTION**, then that value is, unless otherwise specified, {of such a mode that it is} acceptable to {2.1.3.6.d} **'MOID'**.

{This rule makes it possible, in the semantics, to discuss yields without explicitly prescribing their modes.}

c) If the elaboration of some construct A in some environ E is not otherwise specified in the semantics of this Report, and if B is the only direct descendent of A which needs elaboration {see below}, then the elaboration of A in E consists of the elaboration of B in E and the yield, if any, of A is the yield, if any, of B {; this automatic elaboration is termed the "pre-elaboration" of A in E}.

A construct needs no elaboration if it is invisible {1.1.3.2.h}, if it is a **symbol** {9.1.1.h}, or if its elaboration is not otherwise specified in the semantics of this Report and none of its direct descendents needs elaboration.

{Thus the elaboration of the **reference-to-real-closed-clause** (3.1.1.a) *(x := 3.14)* is (and yields the same value as) the elaboration of its constituent **reference-to-real-serial-clause** (3.2.1.a) *x := 3.14*.}

2.1.4.2. Serial and collateral actions

a) An action may be "inseparable", "serial" or "collateral". A serial or collateral action consists of one or more other actions, termed its "direct actions". An inseparable action does not consist of other actions {; what actions are inseparable is left undefined by this Report}.

b) A "descendent action" of another action B is a direct action either of B, or of a descendent action of B.

c) An action A is the "direct parent" of an action B if B is a direct action {a} of A.

d) The direct actions of a serial action S take place one after the other; i.e., the completion {2.1.4.3.c,d} of a direct action of S is followed by the initiation {2.1.4.3.b,c} of the next direct action, if any, of S. {The elaboration of a scene, being in general composed of a sequence of actions, is a serial action.}

e) The direct actions of a collateral action are merged in time; i.e., one of its descendent inseparable actions which, at that moment, is "active" {2.1.4.3.a} is chosen and carried out, upon the completion {2.1.4.3.c} of which another such action is chosen, and so on {until all are completed}.

If two actions {collateral with each other} have been said to be "incompatible with" {10.2.4} each other, then {they shall not be merged; i.e.,} no descendent inseparable action of the one shall (then the one {if it is already inseparable} shall not) be chosen if, at that moment, the other is active and one or more, but not all, of its descendent inseparable actions have already been completed; otherwise, the method of choice is left undefined in this Report.

f) If one or more scenes are to be "elaborated collaterally", then this elaboration is the collateral action consisting of the {merged} elaboration of those scenes.

2.1.4.3. Initiation, completion and termination

a) An action is either "active" or "inactive".
An action becomes active when it is "initiated" {b,c} or "resumed" {g} and it becomes inactive when it is "completed" {c,d}, "terminated" {e}, "halted" {f} or "interrupted" {h}.

b) When a serial action is "initiated", then the first of its direct actions is initiated. When a collateral action is "initiated", then all of its direct actions are initiated.

c) When an inseparable action is "initiated", it may then be carried out {see 2.1.4.2.e}, whereupon it is "completed".

d) A serial action is "completed" when its last direct action has been completed. A collateral action is "completed" when all of its direct actions have been completed.

e) When an action A {whether serial or collateral} is "terminated", then all of its direct actions {and hence all of its descendent actions} are terminated {whereupon another action may be initiated in its place}. {Termination of an action is brought about by the elaboration of a **jump** (5.4.4.2).}

f) When an action is "halted", then all of its active direct actions {and hence all of its active descendent actions} are halted. {An action may be halted during a "calling" of the routine yielded by the **operator *down*** (10.2.4.d), whereupon it may subsequently be resumed during a calling of the routine yielded by the **operator *up*** (10.2.4.e).}

If, at any time, some action is halted and it is not descended from a "process" of a "parallel action" {10.2.4} of whose other process(es) there still exist descendent active inseparable actions, then the further elaboration is undefined. {Thus it is not defined that the elaboration of the **collateral-clause** in

```
begin sema sergei = level 0;
   (par begin (down sergei; print (pokrovsky)), skip end,
      (read (pokrovsky); up sergei))
end
```

will ever be completed.}

g) When an action A is "resumed", then those of its direct actions which had been halted, consequent upon the halting of A are resumed.

h) An action may be "interrupted" by an event {e.g., "overflow"} not specified by the semantics of this Report but caused by the computer if its

limitations {2.2.2.b} do not permit satisfactory elaboration. When an action is interrupted, then all of its direct actions, and possibly its direct parent also, are interrupted. {Whether, after an interruption, that action is resumed, some other action is initiated or the elaboration of the **program** ends, is left undefined by this Report.}

{The effect of the definitions given above is as follows:

During the elaboration of a **program** (2.2.2.a) the elaboration of its **closed-clause** in the empty primal environ is active. At any given moment, the elaboration of one scene may have called for the elaboration of some other scene or of several other scenes collaterally. If and when the elaboration of that other scene or scenes has been completed, the next step of the elaboration of the original scene is taken, and so on until it, in turn, is completed.

It will be seen that all this is analogous to the calling of one subroutine by another; upon the completion of the execution of the called subroutine, the execution of the calling subroutine is continued; the semantic rules given in this Report for the elaboration of the various paranotions correspond to the texts of the subroutines; the semantic rules may even, in suitable circumstances, invoke themselves recursively (but with a different construct or in a different environ on each occasion).

Thus there exists, at each moment, a tree of active actions descended (2.1.4.2.b) from the elaboration of the **program**.}

### 2.1.5. Abbreviations

{In order to avoid some long and turgid phrases which would otherwise have been necessary in the Semantics, certain abbreviations are used freely throughout the text of this Report.}

a) The phrase "the A of B", where A and B are paranotions, stands for "the A which is a direct descendent {1.1.3.2.f} of B".

{This permits the abbreviation of "direct descendent of" to "of" or "its", e.g., in the **assignation** (5.2.1.1.a) *i* := *1*, *i* is "its" **destination** (or *i* is the, or a, **destination** "of" the **assignation** *i* := *1*), whereas *i* is not a **destination** of the **serial-clause** *i* := *1*; *j* := *2* (although it is a constituent **destination** (1.1.4.2.d) of it).}

b) The phrase "C in E", where C is a construct and E is an environ, stands for "the scene composed {2.1.1.1.d} of C and E". It is sometimes even further shortened to just "C" when it is clear which environ is meant.

{Since the process of elaboration (2.1.4.1.a) may be applied only to scenes, this abbreviation appears most frequently in forms such as "A **loop-clause** C, in an environ E1, is elaborated ... " (3.5.2) and "An **assignation** A is elaborated ... " (5.2.1.2.a, where it is the elaboration of A in any appropriate environ that is being discussed).}

c) The phrase "the yield of S", where S is a scene whose elaboration is not explicitly prescribed, stands for "the yield obtained by initiating the elaboration of S and awaiting its completion".

{Thus the sentence (3.2.2.c):

"W is the yield of that **unit**;"

(which also makes use of the abbreviation defined in b above) is to be interpreted as meaning:

"W is the yield obtained upon the completion of the elaboration, hereby initiated, of the scene composed of that **unit** and the environ under discussion;" .}

d) The phrase "the yields of $S_1$, ... , $S_n$", where $S_1$, ... , $S_n$ are scenes whose elaboration is not explicitly prescribed, stands for "the yields obtained by initiating the collateral elaboration {2.1.4.2.f} of $S_1$, ... , $S_n$ and awaiting its completion {which implies the completion of the elaboration of them all}".

If some or all of $S_1$, ... , $S_n$ are described as being, in some environ, certain constituents of some construct, then their yields are to be considered as being taken in the textual order {1.1.3.2.i} of those constituents within that construct.

{Thus the sentence (3.3.2.b):

"let $V_1$, ... , $V_m$ be the {collateral} yields of the constituent **units** of C;"

is to be interpreted as meaning:

"let $V_1$, ... , $V_m$ be the respective yields obtained upon the completion of the collateral elaboration, hereby initiated, of the scenes composed of the constituent **units** of C, considered in their textual order, together with the environ in which C was being elaborated;" .}

e) The phrase "if A is B", where A and B are hypernotions, stands for "if A is equivalent {2.1.1.2.a} to B".

{Thus, in "Case C: **'CHOICE'** is some **'choice using UNITED'**" (3.4.2.b), it matters not whether **'CHOICE'** happens to begin with **'choice using union of'** or with some **'choice using MU definition of union of'**.}

f) The phrase "the mode is A", where A is a hypernotion, stands for "the mode {is a class of **'MOID**'s which} includes A".

{This permits such shortened forms as "the mode is some **'structured with FIELDS mode'**", "the mode begins with **'union of'**", and "the mode envelops a **'FIELD'**"; in general, a mode may be specified by quoting just one of the **'MOID**'s included in it.}

g) The phrase "the value selected (generated) by the **field-selector** F" stands for "if F is a **field-selector-with-TAG** {4.8.1.f}, then the value selected {2.1.3.3.a,e, 2.1.3.4.k} (generated {2.1.3.4.l}) by that **'TAG'**".

## 2.2. The program

### 2.2.1. Syntax

a) **program : strong void new closed clause**{31a}.
{See also 10.1.}

### 2.2.2. 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 its **strong-void-new-closed-clause** in an empty environ {2.1.1.1.c} termed the "primal environ".

{Although the purpose of this Report is to define the meaning of a **particular-program** {10.1.1.g}, that meaning is established only by first defining the meaning of a **program** in which that **particular-program** is embedded (10.1.2).}

{In this Report, the syntax says which sequences of symbols are terminal productions of '**program**', 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 (see also 1.1.3.2.f), this syntactic ambiguity does not lead to a semantic ambiguity.}

b) In ALGOL 68, a specific syntax for constructs is provided which, together with its recursive definition, makes it possible to describe and to distinguish between arbitrarily large production trees, to distinguish between arbitrarily many different values of a given mode (except certain modes like '**boolean**' and '**void**') 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 this Report nor that it has the same possibilities. It is not assumed that these two notations are the same nor 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 computer can handle arbitrary amounts of presented information. It is not assumed that the speed of the computer is sufficient to elaborate a given **program** within a prescribed lapse of time, nor that the number of objects and relationships that can be established is sufficient to elaborate it at all.

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

also **particular-programs** of a language B, and if each such **particular-program** for which a meaning is defined in A has the same defined meaning in B, then A is said to be a "sublanguage" of B, and B a "superlanguage" of A.

{Thus a sublanguage of ALGOL 68 might be defined by omitting some part of the syntax, by omitting some part of the **standard-prelude**, and/or by leaving undefined something which is defined in this Report, so as to enable more efficient solutions to certain classes of problem or to permit implementation on smaller machines.

Likewise, a superlanguage of ALGOL 68 might be defined by additions to the syntax, semantics or **standard-prelude**, so as to improve efficiency (by allowing the user to provide additional information) or to permit the solution of problems not readily amenable to ALGOL 68.}

A model is said to be an implementation of a sublanguage if it does not restrict the use of the sublanguage in other respects than those mentioned above.

{See 9.3.c for the term "implementation of the reference language".}

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

{In an implementation, the **particular-program** may be "compiled", i.e., translated into an "object program" in the code of the physical machine. Under 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 **applied-indicators** which identify **defining-indicators** not contained in that part, then compilation into an efficient object program may be assured by preceding the part by a sequence of **declarations** containing those **defining-indicators**.}

{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. See, however, 9.2 for the suggested use of **pragmats** to control such actions.}

# PART II

## Fundamental Constructions

{This part presents the essential structure of **programs**:

- the general rules for constructing them;

• the ways of defining **indicators** and their properties, at each new level of construction;
• the constructs available for programming primitive actions.}

## 3. Clauses

{Clauses provide
• a hierarchical structure for **programs**,
• the introduction of new **ranges** of definitions,
• serial or collateral composition, parallelism, choices and loops.}

### 3.0.1. Syntax

a) ***phrase : SOME unit{32d) ; NEST declaration of DECS{41a).**
b) ***SORT MODE expression : SORT MODE NEST UNIT{5A).**
c) ***statement : strong void NEST UNIT{5A).**
d) ***MOID constant : MOID NEST DEFIED identifier with TAG{48a,b) ;**
**MOID NEST denoter{80a).**
e) ***MODE variable :**
**reference to MODE NEST DEFIED identifier with TAG{48a,b).**
f) ***NEST range : SOID NEST serial clause defining LAYER{32a) ;**
**SOID NEST chooser CHOICE STYLE clause{34b) ;**
**SOID NEST case part of choice using UNITED{34i) ;**
**NEST STYLE repeating part with DEC{35e) ;**
**NEST STYLE while do part{35f) ;**
**PROCEDURE NEST routine text{541a,b).**

{**NEST-ranges** arise in the definition of "identification" (7.2.2.b).}

### 3.0.2. Semantics

A "nest" is a **'NEST'**. The nest "of" a construct is the **'NEST'** enveloped by the original of that construct, but not by any **'defining LAYER'** contained in that original.

{The nest of a construct carries a record of all the **declarations** forming the environment in which that construct is to be interpreted.
Those constructs which are contained in a **range** R, but not in any smaller **range** contained within R, may be said to comprise a "reach". All constructs in a given reach have the same nest, which is that of the immediately surrounding reach with the addition of one extra **'LAYER'**. The syntax ensures (3.2.1.b, 3.4.1.i,j,k, 3.5.1.e, 5.4.1.1.b) that each **'PROP'** (4.8.1.E) or "property" in the extra **'LAYER'** is matched by a **defining-indicator** (4.8.1.a) contained in a **definition** in that reach.}

## 3.1. Closed clauses

{**Closed-clauses** are usually used to construct **units** from **serial-clauses** as, e.g.,
*(**real** x; read (x); x)* in
*(**real** x; read (x); x) + 3.14.*}

3.1.1. Syntax

A) **SOID :: SORT MOID.**
B) **PACK :: STYLE pack.**

a) **SOID NEST closed clause**{22a,5D,551a,A341h,A349a} :
**SOID NEST serial clause defining LAYER**{32a} **PACK.**
{**LAYER :: new DECSETY LABSETY.**}

{Example:

a) ***begin*** *x:=1; y:=2* ***end*** }

{The yield of a **closed-clause** is that of its constituent **serial-clause**, by way of pre-elaboration (2.1.4.1.c).}

3.2. Serial clauses

{The purposes of **serial-clauses** are
- the construction of new **ranges** of definitions, and
- the serial composition of actions.

A **serial-clause** consists of a possibly empty sequence of unlabelled **phrases**, the last of which, if any, is a **declaration**, followed by a sequence of possibly labelled **units**. The **phrases** and the **units** are separated by **go-on-tokens**, viz., semicolons. Some of the **units** may instead be separated by **completers**, viz., ***exit***s; after a **completer**, the next **unit** must be labelled so that it can be reached. The value of the final **unit**, or of a **unit** preceding an ***exit***, determines the value of the **serial-clause**.

For example, the following **serial-clause** yields true if and only if the vector *a* contains the integer 8:

***int*** *n; read (n);*
*[1 : n]* ***int*** *a; read (a);*
***for*** *i* ***to*** *n* ***do if*** *a [i] = 8* ***then goto*** *success* ***fi od;***
***false exit***
*success:* ***true*** .}

3.2.1. Syntax

a) **SOID NEST serial clause defining new PROPSETY**{31a,34f,1,35h} :
**SOID NEST new PROPSETY series with PROPSETY**{b}.
{Here **PROPSETY :: DECSETY LABSETY.**}
b) **SOID NEST series with PROPSETY**{a,b,34c} :
**strong void NEST unit**{d}, **go on**{94f} **token,**
**SOID NEST series with PROPSETY**{b} ;
**where (PROPSETY) is (DECS DECSETY LABSETY),**
**NEST declaration of DECS**{41a}, **go on**{94f} **token,**
**SOID NEST series with DECSETY LABSETY**{b} ;
**where (PROPSETY) is (LAB LABSETY),**
**NEST label definition of LAB**{c},
**SOID NEST series with LABSETY**{b} ;

**where (PROPSETY) is (LAB LABSETY)**
**and SOID balances SOID1 and SOID2{e), SOID1 NEST unit{d),**
**completion{94f} token, NEST label definition of LAB{c),**
**SOID2 NEST series with LABSETY{b} ;**
**where (PROPSETY) is (EMPTY),**
**SOID NEST unit{d).**

c) **NEST label definition of label TAG{b} :**
**label NEST defining identifier with TAG{48a), label{94f} token.**

d) **SOME unit{b,33b,g,34i,35d,46m,n,521c,532e,541a,b,543c,**
A34Ab,c,d} : **SOME UNIT{5A,-).**

e) **WHETHER SORT MOID balances**
**SORT1 MOID1 and SORT2 MOID2{b,33b,34d,h} :**
**WHETHER SORT balances SORT1 and SORT2{f}**
**and MOID balances MOID1 and MOID2{g}.**

f) **WHETHER SORT balances SORT1 and SORT2{e,522a} :**
**where (SORT1) is (strong), WHETHER (SORT2) is (SORT) ;**
**where (SORT2) is (strong), WHETHER (SORT1) is (SORT).**

g) **WHETHER MOID balances MOID1 and MOID2{e} :**
**where (MOID1) is (MOID2), WHETHER (MOID) is (MOID1) ;**
**where (MOID1) is (transient MOID2),**
**WHETHER (MOID) is (MOID1) ;**
**where (MOID2) is (transient MOID1),**
**WHETHER (MOID) is (MOID2).**

h) ***SOID unitary clause : SOID NEST unit{d).**

i) ***establishing clause :**
**SOID NEST serial clause defining LAYER{32a) ;**
**MODE NEST enquiry clause defining LAYER{34c).**

{Examples:

b)
```
read (x1); real s := 0;
  sum: for i to n do (x1 [i] > 0 | s +:= x1 [i] | nonpos) od exit
  nonpos: print (s) •
real s := 0;
  sum: for i to n do (x1 [i] > 0 | s +:= x1 [i] | nonpos) od exit
  nonpos: print (s) •
sum: for i to n do (x1 [i] > 0 | s +:= x1 [i] | nonpos) od exit
  nonpos: print (s) •
for i to n do (x1 [i] > 0 | s +:= x1 [i] | nonpos) od exit
  nonpos: print (s) •
print (s)
```

c) *sum:* d) *print (s)* }

{Often, a **series** must be "balanced" (3.2.1.e). For remarks concerning balancing, see 3.4.1.}

3.2.2. Semantics

a) The yield of a **serial-clause**, in an environ E, is the yield of the elaboration of its **series**, or of any **series** elaborated "in its place" {5.4.4.2}, in the environ "established" {b} around E according to that **serial-clause**; it is required that the yield be not newer in scope than E.

b) The environ E "established"

- upon an environ E1, possibly not specified, {which determines its scope,}
- around an environ E2 {which determines its composition},
- according to a **NOTION-defining-new-PROPSETY** C, possibly absent, {which prescribes its locale,}
- with values $V_1$, ... , $V_n$, possibly absent, {which are possibly to be ascribed,}

is determined as follows:

- if E1 is not specified, then let E1 be E2;
- E is newer in scope than E1 and is composed of E2 and a new locale corresponding to **'PROPSETY'**, if C is present, and to **'EMPTY'** otherwise;

Case A: C is an **establishing-clause**:

For each constituent **mode-definition** M, if any, of C,

- the scene composed of

(i) the **actual-declarer** of M, and

(ii) the environ necessary for {7.2.2.c} that **actual-declarer** in E,

is ascribed in E to the **mode-indication** of M;

For each constituent **label-definition** L, if any, of C,

- the scene composed of

(i) the **series** of which L is a direct descendent, and

(ii) the environ E,

is ascribed in E to the **label-identifier** of L;

If each **'PROP'** enveloped by **'PROPSETY'** is some **'DYADIC TAD'** or **'label TAG'**,

then E is said to be "nonlocal" {see 5.2.3.2.b};

Case B: C is a **declarative**, a **for-part** or a **specification**:

For i = 1, ... , n, where n is the number of **'DEC'**s enveloped by **'PROPSETY'**,

- $V_i$ is ascribed {4.8.2.a} in E to the i-th constituent **defining-identifier**, if any, of C and, otherwise {in the case of an invisible **for-part**}, to an **integral-defining-indicator-with-letter-aleph**;

If C is a **for-part** or a **specification**,

then E is nonlocal.

{Other cases, i.e., when C is absent:

- E is local (see 5.2.3.2.b), but not further defined.}

c) The yield W of a **series** C is determined as follows:

If C contains a direct descendent **unit** which is not followed by a **go-on-token**,

then

- W is the yield of that **unit**;

otherwise,

- the **declaration** or the **unit**, if any, of C is elaborated;
- W is the yield of the **series** of C.

{See also 5.4.4.2.Case A.}

3.3. Collateral and parallel clauses

{**Collateral-clauses** allow an arbitrary merging of streams of actions. **Parallel-clauses** provide, moreover, levels of coordination for the synchronization (10.2.4) of that merging.

A **collateral-** or **parallel-clause** consists of a sequence of **units** separated by **and-also-symbols** (viz., ","), and is enclosed by parentheses or by a ***begin-end*** pair; a **parallel-clause** begins moreover with ***par.***

**Collateral-clauses**, but not **parallel-clauses**, may yield stowed values composed from the yields of the constituent **units**.

Examples of **collateral-clauses** yielding stowed values:

```
[ ] int q = (1, 4, 9, 16, 25);
struct (int price, string category) bike := (150, "sport").
```

Example of a **parallel-clause** which synchronizes eating and speaking:

```
proc void eat, speak; sema mouth = level 1;
par begin
    do
        down mouth;
        eat;
        up mouth
    od,
    do
        down mouth;
        speak;
        up mouth
    od
end .}
```

3.3.1. Syntax

a) **strong void NEST collateral clause**{5D,551a} :
**strong void NEST joined portrait**{b} **PACK.**

b) **SOID NEST joined portrait**{a,b,c,d,34g} :
**where SOID balances SOID1 and SOID2**{32e},
**SOID1 NEST unit**{32d}, **and also**{94f} **token,**
**SOID2 NEST unit**{32d}
**or alternatively SOID2 NEST joined portrait**{b}.

c) **strong void NEST parallel clause**{5D,551a} :
**parallel**{94f} **token, strong void NEST joined portrait**{b} **PACK.**

d) **strong ROWS of MODE NEST collateral clause**{5D,551a} :
**where (ROWS) is (row),**
**strong MODE NEST joined portrait**{b} **PACK ;**

**where (ROWS) is (row ROWS1),**
**strong ROWS1 of MODE NEST joined portrait{b} PACK ;**
**EMPTY PACK.**

e) **strong structured with**
**FIELDS FIELD mode NEST collateral clause{5D,551a} :**
**NEST FIELDS FIELD portrait{f} PACK.**

f) **NEST FIELDS FIELD portrait{e,f} :**
**NEST FIELDS portrait{f,g}, and also{94f} token,**
**NEST FIELD portrait{g}.**
**{FIELD :: MODE field TAG.}**

g) **NEST MODE field TAG portrait{f} : strong MODE NEST unit{32d).**

h) *** structure display : strong structured with**
**FIELDS FIELD mode NEST collateral clause{e).**

i) *** row display : strong ROWS of MODE NEST collateral clause{d).**

j) *** display : strong STOWED NEST collateral clause{d,e).**

k) *** vacuum : EMPTY PACK.**

{Examples:

a) *(x := 1, y := 2)* b) *x := 1, y := 2*
c) ***par** (task1, task2)* d) *(1, 2)* (in [ ] ***real** (1, 2))*
e) *(1, 2)* (in ***compl** (1, 2))* f) *1, 2*
g) *1* }

{**Structure-displays** must contain at least two **FIELD-portraits**, for, otherwise, in the reach of

***mode m** = **struct** (**ref m** m); **m** nobuo, yoneda;,*

the **assignation** *nobuo := (yoneda)* would be syntactically ambiguous and could produce different effects; however, *m **of** nobuo := yoneda* is unambiguous.

**Row-displays** contain zero, two or more constituent **units**. It is also possible to present a single value as a multiple value, e.g., *[1: 1] **int** v := 123*, but this uses a coercion known as rowing (6.6).}

3.3.2. Semantics

a) The elaboration of a **void-collateral-clause** or **void-parallel-clause** consists of the collateral elaboration of its constituent **units** and yields empty.

b) The yield W of a **STOWED-collateral-clause** C is determined as follows:
If the direct descendent of C is a **vacuum**,
then {**'STOWED'** is some **'ROWS of MODE'** and} each bound pair in the descriptor of W is (1, 0) {and it has one ghost element whose value is irrelevant};
otherwise,

- let $V_1$, ... , $V_m$ be the {collateral} yields of the constituent **units** of C;

Case A: **'STOWED'** is some **'structured with FIELDS mode'**:

- the fields of W, taken in order, are $V_1$, ... , $V_m$;

Case B: **'STOWED'** is some **'row of MODE1'**:

- W is composed of

(i) a descriptor ((1,m)),

(ii) $V_1$, ... , $V_m$;

For i = 1, ... , m,

- the element selected by the index (i) in W is $V_i$;

Case C: **'STOWED'** is some **'row ROWS of MODE2'**:

- it is required that the descriptors of $V_1$, ... , $V_m$ be identical;
- let the descriptor of {say} $V_1$ be $((l_1, u_1), \ldots, (l_n, u_n))$;
- W is composed of

(i) a descriptor $((1, m), (l_1, u_1), \ldots, (l_n, u_n))$;

(ii) the elements of $V_1$, ... , $V_m$;

For i = 1, ... , m,

- the element selected by an index $(i, i_1, \ldots, i_n)$ in W is that selected by $(i_1, \ldots, i_n)$ in $V_i$.

{Note that in [ , , ] ***char*** *block = ("abc", "def")*, the descriptor of the three-dimensional yield W will be ((1, 2), (1, 1), (1, 3)), since the **units** *"abc"* and *"def"* are first rowed (6.6), so that $V_1$ and $V_2$ have descriptors ((1, 1), (1, 3)).}

3.4. Choice clauses

{**Choice-clauses** enable a dynamic choice to be made among different paths in a computation. The choice among the alternatives (the **in-CHOICE-** and the **out-CHOICE-clause**) is determined by the success or failure of a test on a truth value, on an integer or on a mode. The value under test is computed by an **enquiry-clause** before the choice is made.

A **choice-using-boolean-clause** (or **conditional-clause**) is of the form

*( x > 0 | x | 0 )* in the "brief" style, or

***if*** *x > 0* ***then*** *x* ***else*** *0* ***fi*** in the "bold" style;

*x > 0* is the **enquiry-clause**, ***then*** *x* is the **in-CHOICE-clause** and ***else*** *0* is the **out-CHOICE-clause**; all three may have the syntactical structure of a **series**, because all **choice-clauses** are well closed. A **choice-using-boolean-clause** may also be reduced to

*( x < 0 | x := −x )* or

***if*** *x < 0* ***then*** *x := −x* ***fi***;

the omitted **out-CHOICE-clause** is then understood to be an ***else skip***. On the other hand, the choice can be reiterated by writing

*( x > 0 | 1 + x |: x < 0 | 1 − x | 1 )* or

***if*** *x > 0* ***then*** *1 + x* ***elif*** *x < 0* ***then*** *1 − x* ***else*** *1* ***fi***,

and so on; this is to be understood as
*( x > 0 | 1 + x | ( x < 0 | 1 − x | 1 )).*

**CASE-clauses**, which define choices depending on an integer or on a mode, are different in that the **in-CASE-clause** is further decomposed into **units**. The general pattern is
*( --- | --- , ... , --- | --- )* or
***case*** *---* ***in*** *---* *, ... ,* *---* ***out*** *---* ***esac***.
The choice may also be reiterated by use of ***ouse***.

In a **choice-using-integral-clause** (or **case-clause**), the parts are simply **units** and there must be at least two of them; the choice among the **units** follows their textual ordering.
Example:

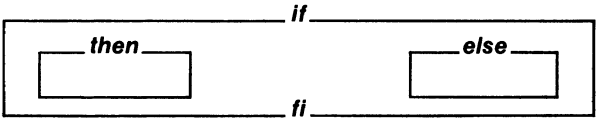
```
proc void work, relax, enjoy;
case int day; read (day); day
in work, work, work, work, work, relax, enjoy
out print ((day, "is not in the week"))
esac.
```

In a **choice-using-UNITED-clause** (or **conformity-clause**), which tests modes, each **case-part-of-CHOICE** is of the form *(**declarer** identifier)*: *unit* or *(**declarer**)*: *unit*. The mode specified by the **declarer** is compared with the mode of the value under test; the **identifier**, if present, is available inside the **unit** to access that value, with the full security of syntactical mode checking. The **'UNITED'** mode provides the required freedom for the mode of the value under test; moreover, that **'UNITED'** mode must contain the mode of each **specification** for, otherwise, the corresponding **case-part-of-CHOICE** could never be chosen.
Example:

```
mode boy = struct (int age, real weight),
mode girl = struct (int age, real beauty);
proc union (boy, girl) newborn;
case newborn in
    (boy john): print (weight of john),
    (girl mary): print (beauty of mary)
esac.)
```

(The flowers that bloom in the spring,
Tra la,
Have nothing to do with the case.
Mikado, W.S. Gilbert.)

(The hierarchy of **ranges** in **conditional-clauses** is illustrated by



and similarly for the other kinds of choice. Thus the nest and the environ of the **enquiry-clause** remain valid over the **in-CHOICE-clause** and the **out-CHOICE-clause**. However, no transfer back from the **in-** or **out-CHOICE-clause** into the **enquiry-clause** is possible, since the latter can contain no **label-definitions** (except within a **closed-clause** contained within it).]

3.4.1. Syntax

A) **CHOICE :: choice using boolean ; CASE.**

B) **CASE :: choice using integral ; choice using UNITED.**

a) **SOID NEST1 CHOICE clause{5D,551a,A341h,A349a} :**
**CHOICE STYLE start{91a,-},**
**SOID NEST1 chooser CHOICE STYLE clause{b},**
**CHOICE STYLE finish{91e,-}.**

b) **SOID NEST1 chooser choice using MODE STYLE clause{a,l} :**
**MODE NEST1 enquiry clause defining LAYER2{c,-},**
**SOID NEST1 LAYER2 alternate choice using MODE STYLE clause{d}.**

c) **MODE NEST1 enquiry clause defining new DECSETY2{b,35g} :**
**meek MODE NEST1 new DECSETY2 series with DECSETY2{32b}.**

d) **SOID NEST2 alternate CHOICE STYLE clause{b} :**
**SOID NEST2 in CHOICE STYLE clause{e} ;**
**where SOID balances SOID1 and SOID2{32e},**
**SOID1 NEST2 in CHOICE STYLE clause{e},**
**SOID2 NEST2 out CHOICE STYLE clause{l}.**

e) **SOID NEST2 in CHOICE STYLE clause{d} :**
**CHOICE STYLE in{91b,-}, SOID NEST2 in part of CHOICE{f,g,h}.**

f) **SOID NEST2 in part of choice using boolean{e} :**
**SOID NEST2 serial clause defining LAYER3{32a}.**

g) **SOID NEST2 in part of choice using integral{e} :**
**SOID NEST2 joined portrait{33b}.**

h) **SOID NEST2 in part of choice using UNITED{e,h} :**
**SOID NEST2 case part of choice using UNITED{i} ;**
**where SOID balances SOID1 and SOID2{32e},**
**SOID1 NEST2 case part of choice using UNITED{i},**
**and also{94f} token,**
**SOID2 NEST2 in part of choice using UNITED{h}.**

i) **SOID NEST2 case part of choice using UNITED{h} :**
**MOID NEST2 LAYER3 specification defining LAYER3{j,k,-},**
**where MOID unites to UNITED{64b},**
**SOID NEST2 LAYER3 unit{32d}.**

[Here **LAYER3 :: new MODE TAG ; new EMPTY.**]

j) **MODE NEST3 specification defining new MODE TAG3{i} :**
**NEST3 declarative defining new MODE TAG3{541e} brief pack, colon{94f} token.**

k) **MOID NEST3 specification defining new EMPTY{i} :**
**formal MOID NEST3 declarer{46b} brief pack, colon{94f} token.**

l) **SOID NEST2 out CHOICE STYLE clause**{d} :
**CHOICE STYLE out**{91d,-},
**SOID NEST2 serial clause defining LAYER3**{32a} ;
**CHOICE STYLE again**{91c,-},
**SOID NEST2 chooser CHOICE2 STYLE clause**{b},
**where CHOICE2 may follow CHOICE**{m}.

m) **WHETHER choice using MODE2 may follow**
**choice using MODE1**{l} :
**where (MODE1) is (MOOD), WHETHER (MODE2) is (MODE1)** ;
**where (MODE1) begins with (union of),**
**WHETHER (MODE2) begins with (union of).**

n) ***SOME choice clause : SOME CHOICE clause**{a}.
o) ***SOME conditional clause : SOME choice using boolean clause**{a}.
p) ***SOME case clause : SOME choice using integral clause**{a}.
q) ***SOME conformity clause : SOME choice using UNITED clause**{a}.

{Examples:

a) *(x > 0 | x | 0)* •
***case** i **in** princeton, grenoble **out** finish **esac** •*
***case** uir **in (int** i**)**: print (i), **(real)**: print ("no") **esac***
b) *x > 0 | x | 0*
c) *x > 0 • i • uir*
d) *| x • | x | 0*
e) *| x •*
***in** princeton, grenoble •*
***in (int** i**)**: print (i), **(real)**: print ("no")*
f) *x*
g) *princeton, grenoble*
h) ***(int** i**)**: print (i), **(real)**: print ("no")*
i) ***(int** i**)**: print (i)*
j) ***(int** i**)**:*
k) ***(real)**:*
l) ***out** finish • |: x < 0 | −x | 0* }

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

{Rule d illustrates why '**SORT MOID**'s should be "balanced".If an **alternate-CHOICE-clause** is, say, firm, then at least its **in-CHOICE-clause** or its **out-CHOICE-clause** must be firm, while the other may be strong. For example, in *(p | x | **skip**) + (p | **skip** | y)*, the **conditional-clause** *(p | x | **skip**)* is balanced by making *| x* firm and *| **skip*** strong whereas *(p | **skip** | y)* is balanced by making *| **skip*** strong and *| y* firm. The counterexample *(p | **skip** | **skip**) + y* illustrates that not both may be strong, for otherwise the **operator** + could not be identified.}

3.4.2. Semantics

a) The yield W of a **chooser-CHOICE-clause** C, in an environ E1, is determined as follows:

• let E2 be the environ established {3.2.2.b} around E1 according to the **enquiry-clause** of C;
• let V be the yield, in E2, of that **enquiry-clause**;
• W is the yield of the scene "chosen" (b) by V from C in E2; it is required that W be not newer in scope than E1.

b) The scene S "chosen" by a value V from a **MOID-chooser-CHOICE-clause** C, in an environ E2, is determined as follows:
Case A: '**CHOICE**' is '**choice using boolean**' and V is true:
• S is the constituent **in-CHOICE-clause** of C, in E2;
Case B: '**CHOICE**' is '**choice using integral**' and $1 \leq V \leq n$, where n is the number of constituent **units** of the constituent **in-part-of-CHOICE** of C:
• S is the V-th such **unit**, in E2;
Case C: '**CHOICE**' is some '**choice using UNITED**' and V is acceptable to {2.1.3.6.d} the '**MOID2**' of some constituent **MOID2-specification** D of C {; if there exists more than one such constituent **specification**, it is not defined which one is chosen as D}:
• S is the **unit** following that D, in an environ established {nonlocally (3.2.2.b)} around E2, according to D, with V;
Other Cases {when the **enquiry-clause** has been unsuccessful}:
If C contains a constituent **out-CHOICE-clause** O,
then S is O in E2;
otherwise, S is a **MOID-skip** in E2.

## 3.5. Loop clauses

{**Loop-clauses** are used for repeating dynamically one same sequence of instructions. The number of repetitions is controlled by a finite sequence of equidistant integers, by a condition to be tested each time, or by both.
Example 1:

***int*** *fac* := *1*;
***for*** *i* ***from*** *n* ***by*** *−1* ***to*** *1*
***do*** *fac* ×:= *i* ***od***.

Example 2:

***int*** *a, b; read ((a, b))* ***pr assert*** $a \geq 0 \wedge b > 0$ ***pr***;
***int*** *q* := *0*, *r* := *a*;
***while*** $r \geq b$ ***pr assert*** $a = b \times q + r \wedge 0 \leq r$ ***pr***
***do*** *(q* +:= *1*, *r* −:= *b)* ***od***
***pr assert*** $a = b \times q + r \wedge 0 \leq r \wedge r < b$ ***pr***

(see 9.2 for an explanation of the **pragmats**).

The controlled **identifier**, e.g., *i* in Example 1, is defined over the **repeating-part**. Definitions introduced in the **while-part** are also valid over the **do-part**.

If the controlled **identifier** is not applied in the **repeating-part**, then the **for-part** may be omitted. A **from-part** ***from*** *1* may be omitted; similarly, ***by*** *1* may be omitted. The **to-part** may be omitted if no test on the final

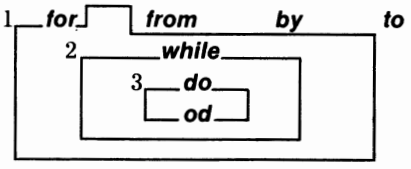
value of the control-integer is required. A **while-part** ***while true*** may be omitted. For example,

***for*** *i* ***from*** *1* ***by*** *1* ***to*** *n* ***while true do*** *print ("a")* ***od***

may be written

***to*** *n* ***do*** *print ("a")* ***od***.

The hierarchy of **ranges** is illustrated by:




]

3.5.1. Syntax

A) **FROBYT :: from ; by ; to.**

a) **strong void NEST1 loop clause{5D,551a} :**
**NEST1 STYLE for part defining new integral TAG2{b},**
**NEST1 STYLE intervals{c},**
**NEST1 STYLE repeating part with integral TAG2{e}.**

b) **NEST1 STYLE for part defining new integral TAG2{a} :**
**STYLE for{94g,-} token,**
**integral NEST1 new integral TAG2 defining identifier**
**with TAG2{48a} ;**
**where (TAG2) is (letter aleph), EMPTY.**

c) **NEST1 STYLE intervals{a} : NEST1 STYLE from part{d} option,**
**NEST1 STYLE by part{d} option,**
**NEST1 STYLE to part{d} option.**

d) **NEST1 STYLE FROBYT part{c} :**
**STYLE FROBYT{94g,-} token, meek integral NEST1 unit{32d}.**

e) **NEST1 STYLE repeating part with DEC2{a} :**
**NEST1 new DEC2 STYLE while do part{f} ;**
**NEST1 new DEC2 STYLE do part{h}.**

f) **NEST2 STYLE while do part{e} :**
**NEST2 STYLE while part defining LAYER3{g},**
**NEST2 LAYER3 STYLE do part{h}.**

g) **NEST2 STYLE while part defining LAYER3{f} :**
**STYLE while{94g,-} token,**
**boolean NEST2 enquiry clause defining LAYER3{34c,-}.**

h) **NEST3 STYLE do part{e,f} :**
**STYLE do{94g,-} token,**
**strong void NEST3 serial clause defining LAYER4{32a},**
**STYLE od{94g,-} token.**

{Examples:

a) ***for*** *i* ***while*** *i* < *n* ***do*** *task1* ***od*** • ***to*** *n* ***do*** *task1; task2* ***od***
b) ***for*** *i* c) ***from*** *−5* ***to*** *+5*

d) ***from*** *−5*
e) ***while*** *i* < *n* ***do*** *task1* ***od*** • ***do*** *task1; task2* ***od***
f) ***while*** *i* < *n* ***do*** *task1; task2* ***od***
g) ***while*** *i* < *n* h) ***do*** *task1; task2* ***od*** }

3.5.2. Semantics

A **loop-clause** C, in an environ E1, is elaborated in the following Steps:

Step 1: All the constituent **FROBYT-parts**, if any, of C are elaborated collaterally in E1;
- let f be the yield of the constituent **from-part**, if any, of C, and be 1 otherwise;
- let b be the yield of the constituent **by-part**, if any, of C, and be 1 otherwise;
- let t be the yield of the constituent **to-part**, if any, of C, and be absent otherwise;
- let E2 be the environ established {nonlocally (3.2.2.b)} around E1, according to the **for-part-defining-new-integral-TAG2** of C, and with the integer f;

Step 2: Let i be the integer accessed {2.1.2.c} by '**integral TAG2**' inside the locale of E2;
If t is not absent,
then
If b > 0 and i > t or if b < 0 and i < t,
then C in E1 {is completed and} yields empty;
{otherwise, Step 3 is taken;}

Step 3: Let an environ E3 and a truth value w be determined as follows:
Case A: C does not contain a constituent **while-part**:
- E3 is E2;
- w is true;

Case B: C contains a constituent **while-part** P:
- E3 is established {perhaps nonlocally (3.2.2.b)} around E2 according to the **enquiry-clause** of P;
- w is the yield in E3 of that **enquiry-clause**;

Step 4: If w is true,
then
- the constituent **do-part** of C is elaborated in E3;
- '**integral TAG2**' is made to access i + b inside the locale of E2;
- Step 2 is taken again;

otherwise,
- C in E1 {is completed and} yields empty.

{The **loop-clause**
***for*** *i* ***from*** *u1* ***by*** *u2* ***to*** *u3* ***while*** *condition* ***do*** *action* ***od***
is thus equivalent to the following **void-closed-clause**:

```
begin int f := u1, int b = u2, t = u3;
    step2:
        if (b > 0 ∧ f ≤ t) ∨ (b < 0 ∧ f ≥ t) ∨ b = 0
        then int i = f;
            if condition
            then action; f +:= b; go to step2
            fi
        fi
end.
```

This equivalence might not hold, of course, if the **loop-clause** contains **local-generators**, or if some of the **operators** above do not identify those in the standard environment(10).}

## 4. Declarations, declarers and indicators

{**Declarations** serve

- to announce new **indicators**, e.g., **identifiers**,
- to define their modes or priorities, and
- to ascribe values to those **indicators** and to initialize **variables**.}

### 4.1. Declarations

#### 4.1.1. Syntax

A) **COMMON :: mode ; priority ; MODINE identity ;**
**reference to MODINE variable ; MODINE operation ;**
**PARAMETER ; MODE FIELDS.**
**{MODINE :: MODE ; routine.}**

a) **NEST declaration of DECS{a,32b} :**
**NEST COMMON declaration of DECS{42a,43a,44a,e,45a,-} ;**
**where (DECS) is (DECS1 DECS2),**
**NEST COMMON declaration of DECS1{42a,43a,44a,e,45a,-},**
**and also{94f} token, NEST declaration of DECS2{a}.**

b) **NEST COMMON joined definition of PROPS PROP**
**{b,42a,43a,44a,e,45a,46e,541e} :**
**NEST COMMON joined definition of PROPS{b,c},**
**and also{94f} token,**
**NEST COMMON joined definition of PROP{c}.**

c) **NEST COMMON joined definition of PROP**
**{b,42a,43a,44a,e,45a,46e,541e} :**
**NEST COMMON definition of PROP{42b,43b,44c,f,45c,46f,541f,-}.**

d) ***definition of PROP : NEST COMMON definition of PROP**
**{42b,43b,44c,f,45c,46f,541f} ;**
**NEST label definition of PROP{32c}.**

{Examples:

a) ***mode r = ref real, s = char •prio*** ∨ *= 2,* ∧ *= 3 •**int** m = 4096 •*
***real** x, y •*
***op*** ∨ *= (**bool** a, b) **bool** : (a | **true** | b)*

b) *r = **ref real**, s = **char** •*∨ *= 2,* ∧ *= 3 •m = 4096 •x, y •*
∨ *= (**bool** a, b) **bool** : (a | **true** | b)*

c) *r = **ref real** •*∨ *= 2 •m = 4096 •x •*
∨ *= (**bool** a, b) **bool** : (a | **true** | b)* }

4.1.2. Semantics

The elaboration of a **declaration** consists of the collateral elaboration of its **COMMON-declaration** and of its **declaration**, if any. {Thus, all the **COMMON-declarations** separated by **and-also-tokens** are elaborated collaterally.}

4.2. Mode declarations

{**Mode-declarations** provide the **defining-mode-indications**, which act as abbreviations for **declarers** constructed from the more primitive ones, or from other **declarers**, or even from themselves.

For example,

***mode array** = [m, n] **real**,* and

***mode book** = **struct** (**string** text, **ref book** next)*

In the latter example, the **applied-mode-indication** ***book*** is not only a convenient abbreviation, but is essential to the **declaration**.}

4.2.1. Syntax

a) **NEST mode declaration of DECS**{41a} :
**mode**{94d} **token, NEST mode joined definition of DECS**{41b,c}.

b) **NEST mode definition of MOID TALLY TAB**{41c} :
**where (TAB) is (bold TAG) or (NEST) is (new LAYER),**
**MOID TALLY NEST defining mode indication with TAB**{48a},
**is defined as**{94d} **token,**
**actual MOID TALLY NEST declarer**{c}.

c) **actual MOID TALLY1 NEST declarer**{b} :
**where (TALLY1) is (i),**
**actual MOID NEST declarator**{46c,d,g,h,o,s,-} ;
**where (TALLY1) is (TALLY2 i),**
**MOID TALLY2 NEST applied mode indication with TAB2**
{48b}.

{Examples:

a) ***mode r = ref real, s = char***
b) ***r = ref real*** c) ***ref real • char*** }

{The use of '**TALLY**' excludes circular chains of **mode-definitions** such as ***mode a = b, b = a***.

**Defining-mode-indications-with-SIZETY-STANDARD** may be declared only in the **standard-prelude**, where the nest is of the form **'new LAYER'** (10.1.1.b).}

4.2.2. Semantics

The elaboration of a **mode-declaration** {involves no action, yields no value and} is completed.

4.3. Priority declarations

{**Priority-declarations** are used to specify the priority of **operators**. Priorities from 1 to 9 are available.

Since **monadic-operators** have effectively only one priority-level, which is higher than that of all **dyadic-operators**, **monadic-operators** do not require **priority-declarations**.}

4.3.1. Syntax

a) **NEST priority declaration of DECS**{41a} :
**priority**{94d} **token, NEST priority joined definition of DECS**{41b,c}.

b) **NEST priority definition of priority PRIO TAD**{41c} :
**priority PRIO NEST defining operator with TAD**{48a},
**is defined as**{94d} **token, DIGIT**{94b} **token,**
**where DIGIT counts PRIO**{c,d}.
{**DIGIT :: digit zero ; digit one ; digit two ; digit three ; digit four ;**
**digit five ; digit six ; digit seven ; digit eight ; digit nine.**}

c) **WHETHER DIGIT1 counts PRIO i**{b,c} :
**WHETHER DIGIT2 counts PRIO**{c,d},
**where (digit one digit two digit three digit four**
**digit five digit six digit seven digit eight digit nine)**
**¬contains (DIGIT2 DIGIT1).**

d) **WHETHER digit one counts i**{b,c} : **WHETHER true.**

{Examples:

a) ***prio*** $\vee = 2, \wedge = 3$ b) $\vee = 2$}

4.3.2. Semantics

The elaboration of a **priority-declaration** {involves no action, yields no value and} is completed.

4.4. Identifier declarations

{**Identifier-declarations** provide **MODE-defining-identifiers**, by means of either **identity-definitions** or **variable-definitions**.

Examples:

***real*** *pi* = *3.1416* •
***real*** *scan* := *0.05*.

The latter example, which is a **variable-declaration**, may be considered as an equivalent form of the **identity-declaration**

***ref real*** *scan* = ***loc real*** := *0.05*.

The elaboration of **identifier-declarations** causes values to be ascribed to their **identifiers**; in the examples given above, 3.1416 is ascribed to *pi* and a new local name which refers to 0.05 is ascribed to *scan*.)

4.4.1. Syntax

A) **MODINE :: MODE ; routine.**
B) **LEAP :: local ; heap ; primal.**

a) **NEST MODINE identity declaration of DECS**{41a} :
**formal MODINE NEST declarer**{b,46b},
**NEST MODINE identity joined definition of DECS**{41b,c}.
b) **VICTAL routine NEST declarer**{a,523b} : **procedure**{94d} **token.**
c) **NEST MODINE identity definition of MODE TAG**{41c} :
**MODE NEST defining identifier with TAG**{48a},
**is defined as**{94d} **token, MODE NEST source for MODINE**{d}.
d) **MODE NEST source for MODINE**{c,f,45c} :
**where (MODINE) is (MODE), MODE NEST source**{521c} ;
**where (MODINE) is (routine), MODE NEST routine text**{541a,b,-}.

e) **NEST reference to MODINE variable declaration of DECS**{41a} :
**reference to MODINE NEST LEAP sample generator**{523b},
**NEST reference to MODINE variable joined**
**definition of DECS**{41b,c}.
f) **NEST reference to MODINE variable definition**
**of reference to MODE TAG**{41c} :
**reference to MODE NEST defining identifier with TAG**{48a},
**becomes**{94c} **token, MODE NEST source for MODINE**{d} ;
**where (MODINE) is (MODE),**
**reference to MODE NEST defining identifier with TAG**{48a}.

g) ***identifier declaration :**
**NEST MODINE identity declaration of DECS**{a} ;
**NEST reference to MODINE variable declaration of DECS**{e}.

{Examples:

a) ***int*** *m* = *4096* • ***proc*** *r10* = ***real*** : *random* × *10*
b) ***proc*** c) *m* = *4096*
d) *4096* • ***real*** : *random* × *10*
e) ***real*** *x*, *y* • ***proc*** *pp* := ***real*** : *random* × *10*
f) *pp* := ***real*** : *random* × *10* • *x* }

4.4.2. Semantics

a) An **identity-declaration** D is elaborated as follows:
• the constituent **sources-for-MODINE** of D are elaborated collaterally;

For each constituent **identity-definition** D1 of D,

- the yield V of the **source-for-MODINE** of D1 is ascribed {4.8.2.a} to the **defining-identifier** of D1.

b) A **variable-declaration** D is elaborated as follows:

- the **sample-generator** {5.2.3.1.b} G of D and all the **sources-for-MODINE**, if any, of the constituent **variable-definitions** of D are elaborated collaterally;

For each constituent **variable-definition-of-reference-to-MODE-TAG** D1 of D,

- let W1 be a "variant" {c}, for '**MODE**', of the value referred to by the yield N of G;
- let N1 be a newly created name equal in scope to N and referring to W1;
- if N1 is a stowed name {2.1.3.2.b}, then N1 is endowed with subnames {2.1.3.3.e, 2.1.3.4.g};
- N1 is ascribed {4.8.2.a} to the **defining-identifier** of D1;
- the yield of the **source-for-MODINE**, if any, of D1 is assigned {5.2.1.2.b} to N1.

{An **actual-declarer** which is common to a number of **variable-definitions** is elaborated only once. For example, the elaboration of

***int*** *m* := *10*; [ *1* : *m* +:= *1* ] ***int*** *p, q; print (m)*

causes 11 to be printed, and not 12; moreover, two new local names referring to multiple values with descriptor ((1, 11)), and undefined elements, are ascribed to *p* and to *q*.}

c) A "variant" of a value V, for a mode M, is a value W acceptable to {2.1.3.6.d} M, and determined as follows:

Case A: M is some '**structured with FIELDS mode**':

For each '**MODE field TAG**' enveloped by '**FIELDS**',

- the field selected by '**TAG**' in W is a variant, for '**MODE**', of the field selected by '**TAG**' in V;

Case B: M is some '**FLEXETY ROWS of MODE1**':

- the descriptor of W is that of V;
- each element of W is a variant, for '**MODE1**', of some element of V;

Other Cases:

- W is any value acceptable to M.

d) The yield of an **actual-routine-declarer** is some routine {whose mode is of no relevance}.

## 4.5. Operation declarations

{**Operation-declarations** provide **defining-operators**.

Example:

***op mc*** = (***real*** *a, b*) ***real*** : *(3 × a < b | a | b)*.

Unlike the case with, e.g., **identifier-declarations**, more than one **operation-declaration** involving the same **TAO-token** may occur in the

same reach; e.g., the previous example may very well be in the same reach as

***op mc*** = (***compl*** *carthy, john*) ***compl*** : (*random* < .5 | *carthy* | *john*);

the **operator** ***mc*** is then said to be "overloaded".}

4.5.1. Syntax

A) **PRAM :: DUO ; MONO.**
B) **TAO :: TAD ; TAM.**

a) **NEST MODINE operation declaration of DECS**{41a} :
**operator**{94d} **token, formal MODINE NEST plan**{b,46p,-},
**NEST MODINE operation joined definition of DECS**{41b,c}.
b) **formal routine NEST plan**{a} : **EMPTY.**
c) **NEST MODINE operation definition of PRAM TAO**{41c} :
**PRAM NEST defining operator with TAO**{48a},
**is defined as**{94d} **token, PRAM NEST source for MODINE**{44d}.

{Examples:

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

4.5.2. Semantics

a) The elaboration of an **operation-declaration** consists of the collateral elaboration of its constituent **operation-definitions**.

b) An **operation-definition** is elaborated by ascribing {4.8.2.a} the routine yielded by its **source-for-MODINE** to its **defining-operator**.

4.6. Declarers

{**Declarers** specify modes. A **declarer** is either a **declarator**, which explicitly constructs a mode, or an **applied-mode-indication**, which stands for some **declarator** by way of a **mode-declaration**. **Declarators** are built from ***void***, ***int***, ***real***, ***bool*** and ***char*** (10.2.2), with the assistance of other symbols such as ***ref***, ***struct***, [ ], ***proc***, and ***union***. For example, ***proc*** (***real***) ***bool*** specifies the mode '**procedure with real parameter yielding boolean**'.

**Actual-declarers**, used typically in **generators**, require the presence of bounds. **Formal-declarers**, used typically in **formal-parameters** and **casts**, are without bounds. The **declarer** following a ***ref*** is always '**virtual**'; it may then specify a '**flexible ROWS of MODE**', because flexibility is a property of names. Since **actual-declarers** follow an implicit '**reference to**' in **generators**, they may also specify '**flexible ROWS of MODE**'.}

4.6.1. Syntax

A) **VICTAL :: VIRACT ; formal.**
B) **VIRACT :: virtual ; actual.**
C) **MOIDS :: MOID ; MOIDS MOID.**

a) **VIRACT MOID NEST declarer{c,e,g,h,523a,b} :**
**VIRACT MOID NEST declarator{c,d,g,h,o,s,-} ;**
**MOID TALLY NEST applied mode indication with TAB{48b,-}.**

b) **formal MOID NEST declarer{e,h,p,r,u,34k,44a,541a,b,e,551a} :**
**where MOID deflexes to MOID{47a,b,c,-},**
**formal MOID NEST declarator{c,d,h,o,s,-} ;**
**MOID1 TALLY NEST applied mode indication with TAB{48b,-},**
**where MOID1 deflexes to MOID{47a,b,c,-}.**

c) **VICTAL reference to MODE NEST declarator{a,b,42c} :**
**reference to{94d} token, virtual MODE NEST declarer{a}.**

d) **VICTAL structured with FIELDS mode NEST declarator{a,b,42c} :**
**structure{94d} token,**
**VICTAL FIELDS NEST portrayer of FIELDS{e} brief pack.**

e) **VICTAL FIELDS NEST portrayer of FIELDS1{d,e} :**
**VICTAL MODE NEST declarer{a,b},**
**NEST MODE FIELDS joined definition of FIELDS1{41b,c} ;**
**where (FIELDS1) is (FIELDS2 FIELDS3),**
**VICTAL MODE NEST declarer{a,b},**
**NEST MODE FIELDS joined definition of FIELDS2{41b,c},**
**and also{94f} token,**
**VICTAL FIELDS NEST portrayer of FIELDS3{e}.**

f) **NEST MODE FIELDS definition of MODE field TAG{41c} :**
**MODE field FIELDS defining field selector with TAG{48c}.**

g) **VIRACT flexible ROWS of MODE NEST declarator{a,42c} :**
**flexible{94d} token, VIRACT ROWS of MODE NEST declarer{a}.**

h) **VICTAL ROWS of MODE NEST declarator{a,b,42c} :**
**VICTAL ROWS NEST rower{i,j,k,l} STYLE bracket,**
**VICTAL MODE NEST declarer{a,b}.**

i) **VICTAL row ROWS NEST rower{h,i} :**
**VICTAL row NEST rower{j,k,l}, and also{94f} token,**
**VICTAL ROWS NEST rower{i,j,k,l}.**

j) **actual row NEST rower{h,i} : NEST lower bound{m}, up to{94f} token,**
**NEST upper bound{n} ; NEST upper bound{n}.**

k) **virtual row NEST rower{h,i} : up to{94f} token option.**

l) **formal row NEST rower{h,i} : up to{94f} token option.**

m) **NEST lower bound{j,532f,g} : meek integral NEST unit{32d}.**

n) **NEST upper bound{j,532f} : meek integral NEST unit{32d}.**

o) **VICTAL PROCEDURE NEST declarator{a,b,42c} :**
**procedure{94d} token, formal PROCEDURE NEST plan{p}.**

p) **formal procedure PARAMETY yielding MOID NEST plan{o,45a} :**
**where (PARAMETY) is (EMPTY), formal MOID NEST declarer{b} ;**
**where (PARAMETY) is (with PARAMETERS),**
**PARAMETERS NEST joined declarer{q,r} brief pack,**
**formal MOID NEST declarer{b}.**

q) **PARAMETERS PARAMETER NEST joined declarer**{p,q} :
**PARAMETERS NEST joined declarer**{q,r}, **and also**{94f} **token**,
**PARAMETER NEST joined declarer**{r}.

r) **MODE parameter NEST joined declarer**{p,q} :
**formal MODE NEST declarer**{b}.

s) **VICTAL union of MOODS1 MOOD1 mode**
**NEST declarator**{a,b,42c} :
**unless EMPTY with MOODS1 MOOD1 incestuous**{47f},
**union of**{94d} **token**,
**MOIDS NEST joined declarer**{t,u} **brief pack**,
**where MOIDS ravels to MOODS2**{47g}
**and safe MOODS1 MOOD1 subset of safe MOODS2**{73l}
**and safe MOODS2 subset of safe MOODS1 MOOD1**{73l,m}.

t) **MOIDS MOID NEST joined declarer**{s,t} :
**MOIDS NEST joined declarer**{t,u}, **and also**{94f} **token**,
**MOID NEST joined declarer**{u}.

u) **MOID NEST joined declarer**{s,t} : **formal MOID NEST declarer**{b}.

{Examples:

a) [*1* : *n*] ***real*** • ***person***
b) [ ] ***real*** • ***string***
c) ***ref real***
d) ***struct*** *(**int** age, **ref person** father, son)*
e) ***ref person*** *father, son* • ***int*** *age, **ref person** father, son*
f) *age*
g) ***flex*** [*1* : *n*] ***real***
h) [*1* : *m*, *1* : *n*] ***real***
i) *1* : *m*, *1* : *n*
j) *1* : *n*
k) :
l) :
m) *1*
n) *n*
o) ***proc*** *(**bool**, **bool**)* ***bool***
p) *(**bool**, **bool**)* ***bool***
q) ***bool***, ***bool***
r) ***bool***
s) ***union*** *(**int**, **char**)*
t) ***int***, ***char***
u) ***int*** }

{For **actual-MOID-TALLY-declarers**, see 4.2.1.c; for **actual-routine-declarers**, see 4.4.1.b.

There are no **declarers** specifying modes such as '**union of integral union of integral real mode mode**' or '**union of integral real integral mode**'. The **declarers** ***union*** *(**int**, **union** (**int**, **real**))* and ***union*** *(**int**, **real**, **int**)* may indeed be written, but in both cases the mode specified is '**union of integral real mode**' (which can as well be spelled '**union of real integral mode**').}

4.6.2. Semantics

a) The yield W of an **actual-MODE-declarer** D, in an environ E, is determined as follows:

If '**MODE**' is some '**STOWED**',
then
- let D1 in E1 be "developed" {c} from D in E;
- W is the yield of {the **declarator**} D1 in an environ established {locally, see 3.2.2.b} upon E and around E1;

otherwise,
- W is any value {acceptable to '**MODE**'}.

b) The yield W of an **actual-STOWED-declarator** D is determined as follows:
Case A: '**STOWED**' is some '**structured with FIELDS mode**':
- the constituent **declarers** of D are elaborated collaterally;
- each field of W is a variant {4.4.2.c}

(i) of the yield of the last constituent **MODE-declarer** of D occurring before the constituent **defining-field-selector** of D selecting {2.1.5.g} that field,
(ii) for that '**MODE**';

Case B: '**STOWED**' is some '**ROWS of MODE**':
- all the constituent **lower-bounds** and **upper-bounds** of D and the **declarer** D1 of D are elaborated collaterally;

For i = 1, ... , n, where n is the number of '**row**'s contained in '**ROWS**',
- let $l_i$ be the yield of the **lower-bound**, if any, of the i-th constituent **row-rower** of D, and be 1 otherwise;
- let $u_i$ be the yield of the **upper-bound** of that **row-rower**;

- W is composed of

(i) a descriptor $((l_1, u_1), \ldots , (l_n, u_n))$,
(ii) variants of the yield of D1, for '**MODE**';

Case C: '**STOWED**' is some '**flexible ROWS of MODE**':
- W is the yield of the **declarer** of D.

c) The scene S "developed from" an **actual-STOWED-declarer** D in an environ E is determined as follows:
If the visible direct descendent D1 of D is a **mode-indication**,
then
- S is the scene developed from that yielded by D1 in E;

otherwise {D1 is a **declarator**},
- S is composed of D1 and E.

d) A given **MOID-declarer** "specifies" the mode '**MOID**'.

## 4.7. Relationships between modes

{Some modes must be deflexed because the mode of a value may not be flexible (2.1.3.6.b). Incestuous unions must be prevented in order to avoid ambiguities. A set of '**UNITED**'s and '**MOODS**'s may be ravelled by replacing all those '**UNITED**'s by their component '**MOODS**'s.}

4.7.1. Syntax

A) **NONSTOWED :: PLAIN ; REF to MODE ; PROCEDURE ; UNITED ; void.**
B) **MOODSETY :: MOODS ; EMPTY.**
C) **MOIDSETY :: MOIDS ; EMPTY.**

a) **WHETHER NONSTOWED deflexes to NONSTOWED {b,e,46b,521c,62a,71n} : WHETHER true.**
b) **WHETHER FLEXETY ROWS of MODE1 deflexes to ROWS of MODE2{b,e,46b,521c,62a,71n} : WHETHER MODE1 deflexes to MODE2{a,b,c,-}.**
c) **WHETHER structured with FIELDS1 mode deflexes to structured with FIELDS2 mode{b,e,46b,521c,62a,71n} : WHETHER FIELDS1 deflexes to FIELDS2{d,e,-}.**
d) **WHETHER FIELDS1 FIELD1 deflexes to FIELDS2 FIELD2{c,d} : WHETHER FIELDS1 deflexes to FIELDS2{d,e,-} and FIELD1 deflexes to FIELD2{e,-}.**
e) **WHETHER MODE1 field TAG deflexes to MODE2 field TAG{c,d} : WHETHER MODE1 deflexes to MODE2{a,b,c,-}.**

f) **WHETHER MOODSETY1 with MOODSETY2 incestuous{f,46s} : where (MOODSETY2) is (MOOD MOODSETY3), WHETHER MOODSETY1 MOOD with MOODSETY3 incestuous{f} or MOOD is firm union of MOODSETY1 MOODSETY3 mode {71m} ; where (MOODSETY2) is (EMPTY), WHETHER false.**

g) **WHETHER MOIDS ravels to MOODS{g,46s} : where (MOIDS) is (MOODS), WHETHER true ; where (MOIDS) is (MOODSETY union of MOODS1 mode MOIDSETY), WHETHER MOODSETY MOODS1 MOIDSETY ravels to MOODS{g}.**

{A component mode of a union may not be firmly coerced to one of the other component modes or to the union of those others (rule f) for, otherwise, ambiguities could arise. For example,

***union (ref int, int) (loc int)***,

is ambiguous in that dereferencing may or may not occur before the uniting. Similarly,

***mode szp = union (szeredi, peter);***
***union (ref szp, szp) (loc szp)***

is ambiguous. Note that, because of ravelling (rule g), the mode specified by the declarer of the **cast** is more closely suggested by ***union (ref szp, szeredi, peter)***.}

## 4.8. Indicators and field selectors

### 4.8.1. Syntax

A) **INDICATOR :: identifier ; mode indication ; operator.**
B) **DEFIED :: defining ; applied.**
C) **PROPSETY :: PROPS ; EMPTY.**
D) **PROPS :: PROP ; PROPS PROP.**
E) **PROP :: DEC ; LAB ; FIELD.**
F) **QUALITY ::**
**MODE ; MOID TALLY ; DYADIC ; label ; MODE field.**
G) **TAX :: TAG ; TAB ; TAD ; TAM.**

a) **QUALITY NEST new PROPSETY1 QUALITY TAX PROPSETY2**
**defining INDICATOR with TAX{32c,35b,42b,43b,44c,f,45c, 541f} :**
**where QUALITY TAX independent PROPSETY1 PROPSETY2**
**{71a,b,c}, TAX{942A,D,F,K} token.**
b) **QUALITY NEST applied INDICATOR with TAX**
**{42c,46a,b,5D,542a,b,544a} :**
**where QUALITY TAX identified in NEST{72a},**
**TAX{942A,D,F,K} token.**
c) **MODE field PROPSETY1 MODE field TAG PROPSETY2 defining**
**field selector with TAG{46f} :**
**where MODE field TAG independent PROPSETY1 PROPSETY2**
**{71a,b,c}, TAG {942A} token.**
d) **MODE field FIELDS applied field selector with TAG{531a} :**
**where MODE field TAG resides in FIELDS{72b,c,-},**
**TAG{942A} token.**

e) ***QUALITY NEST DEFIED indicator with TAX :**
**QUALITY NEST DEFIED INDICATOR with TAX{a,b}.**
f) ***MODE DEFIED field selector with TAG :**
**MODE field FIELDS DEFIED field selector with TAG{c,d}.**

{Examples:

a) *x* (in ***real*** *x, y*) b) *x* (in *x* + *y*)
c) *next* (see 1.1.2) d) *next* (in *next* ***of*** *draft*) }

### 4.8.2. Semantics

a) When a value or a scene V is "ascribed" to a **QUALITY-defining-indicator-with-TAX**, in an environ E, then **'QUALITY TAX'** is made to access V inside the locale of E {2.1.2.c}.

b) The yield W of a **QUALITY-applied-indicator-with-TAX I** in an environ E composed of an environ E1 and a locale L is determined as follows:

If L corresponds to a **'DECSETY LABSETY'** which envelops {1.1.4.1.c} that **'QUALITY TAX'**,
then W is the value or scene, if any, accessed inside L by **'QUALITY TAX'** and, otherwise, is undefined;
otherwise, W is the yield of **I** in E1.

{Consider the following **closed-clause**, which contains another one:

```
begin co range 1 co
    int i = 421, int a := 5, proc p = void : print (a);
    begin co range 2 co
        real a; a := i; p
    end
end.
```

By the time *a* := *i* is encountered during the elaboration, two new environs have been created, one for each range. The **defining-identifier** *i* is first sought in the newer one, E2, is not found there, and then is sought and found in the older one, E1. The locale of E1 corresponds to **'integral letter i reference to integral letter a procedure yielding void letter p'**. The yield of the **applied-identifier** *i* is therefore the value 421 which has been ascribed (a) to **'integral letter i'** inside the locale of E1. The yield of *a*, in *a* := *i*, however, is found from the locale of E2.

When *p* is called (5.4.3.2.b), its **unit** is elaborated in an environ E3 established around E1 but upon E2 (3.2.2.b). This means that, for scope purposes, E3 is newer than E2, but the component environ of E3 is E1. When *a* comes to be printed, it is the yield 5 of the **reference-to-integral-identifier** *a* declared in the outer **range** that is obtained.

Thus, the meaning of an **indicator** applied but not defined within a routine is determined by the context in which the routine was created, rather than that in which it is called.}

## 5. Units

{**Units** are used to program the more primitive actions or to put into one single piece the larger constructs of Chapter 3.

**NOTION-coercees** are the results of coercion (Chapter 6), but **hips** are not; in the case of **ENCLOSED-clauses**, any coercions needed are performed inside them.

The syntax below implies, for example, that *text* **of** *draft* + *"the_end"* is parsed as *(text* **of** *draft)* + *"the_end"* since a **selection** is a **'SECONDARY'** whereas a **formula** is a **'TERTIARY'**.}

### 5.1. Syntax

A) **UNIT**{32d} :: **assignation**{521a} **coercee** ;
**identity relation**{522a} **coercee** ; **routine text**{541a,b} **coercee** ;
**jump**{544a} ; **skip**{552a} ; **TERTIARY**{B}.

B) **TERTIARY**{A,521b,522a} :: **ADIC formula**{542a,b} **coercee** ;
**nihil**{524a} ; **SECONDARY**{C}.

C) **SECONDARY**{B,531a,542c} :: **LEAP generator**{523a} **coercee** ;
**selection**{531a} **coercee** ; **PRIMARY**{D}.

D) **PRIMARY**{C,532a,543a} :: **slice**{532a} **coercee** ; **call**{543a} **coercee** ;
**cast**{551a} **coercee** ; **denoter**{80a} **coercee** ;
**format text**{A341a} **coercee** ;
**applied identifier with TAG**{48b} **coercee** ;
**ENCLOSED clause**{31a,33a,c,d,e,34a,35a}.

{The hyper-rules for '**SORT MOID FORM coercee**' are given in 6.1.1.a, b, c, d and e, the entry rules of the coercion syntax. When the coercion syntax is invoked for some '**SORT MOID FORM coercee**', it will eventually return to a rule in this chapter for some '**MOID1 FORM**' (blind alleys apart). It is the cross-reference to that rule that is given in the metaproduction rules above. No other visible descendent has been produced in the meantime; the coercion syntax merely transforms '**MOID**' into '**MOID1**' for semantical purposes.}

a) ***SOME hip** :
**SOME jump**{544a} ; **SOME skip**{552a} ; **SOME nihil**{524a}.

{The mode of a **hip** is always that required, a posteriori, by its context, and its yield is acceptable to that mode. Since any mode is so easily accommodated, no coercion is permitted.}

## 5.2. Units associated with names

{Names may be assigned to (5.2.1), compared with other names (5.2.2) and created (5.2.3).}

### 5.2.1. Assignations

{In **assignations**, a value is "assigned" to a name. E.g., in *x* := *3.14*, the real number yielded by the **source** *3.14* is assigned to the name yielded by the **destination** *x*.}

#### 5.2.1.1. Syntax

a) **REF to MODE NEST assignation**{5A} :
**REF to MODE NEST destination**{b}, **becomes**{94c} **token**,
**MODE NEST source**{c}.

b) **REF to MODE NEST destination**{a} :
**soft REF to MODE NEST TERTIARY**{5B}.

c) **MODE1 NEST source**{a,44d} : **strong MODE2 NEST unit**{32d},
**where MODE1 deflexes to MODE2**{47a,b,c,-}.

{Examples:

a) *x* := *3.14* b) *x*
c) *3.14* }

5.2.1.2. Semantics

a) An **assignation** A is elaborated as follows:

- let N and W be the {collateral} yields {a name and another value} of the **destination** and **source** of A;
- W is assigned to {b} N;
- the yield of A is N.

b) A value W is "assigned to" a name N, whose mode is some **'REF to MODE'**, as follows:

It is required that

- N be not nil, and that
- W be not newer in scope than N;

Case A: **'MODE'** is some **'structured with FIELDS mode'**:

For each **'TAG'** selecting a field in W,

- that field is assigned to the subname selected by **'TAG'** in N;

Case B: **'MODE'** is some **'ROWS of MODE1'**:

- let V be the {old} value referred to by N;
- it is required that the descriptors of W and V be identical;

For each index **i** selecting an element in W,

- that element is assigned to the subname selected by **i** in N;

Case C: **'MODE'** is some **'flexible ROWS of MODE1'**:

- let V be the {old} value referred to by N;
- N is made to refer to a multiple value composed of

(i) the descriptor of W,

(ii) variants {4.4.2.c} of some element {possibly a ghost element} of V;

- N is endowed with subnames {2.1.3.4.g};

For each index **i** selecting an element in W,

- that element is assigned to the subname selected by **i** in N;

Other Cases {e.g., where **'MODE'** is some **'PLAIN'** or some **'UNITED'**}:

- N is made to refer {2.1.3.2.a} to W.

{Observe how, given

***flex*** [*1* : *0*] [*1* : *3*] ***int*** *flexfix*,

the presence of the ghost element (2.1.3.4.c) ensures that the meaning of *flexfix* := ***loc*** [*1* : *1*] [*1* : *3*] ***int*** is well defined, but that of *flexfix* := ***loc*** [*1* : *1*] [*1* : *4*] ***int*** is not, since the bound pairs of the second dimension are different.}

5.2.2. Identity relations

{**Identity-relations** may be used to ask whether two names of the same mode are the same.

E.g., after the **assignation** *draft* := *("abc", **nil**)*, the **identity-relation** *next **of** draft* :=: ***ref book** (**nil**)* yields true. However, *next **of** draft* :=: ***nil*** yields false because it is equivalent to *next **of** draft* :=: ***ref ref book** (**nil**)*: the yield of *next **of** draft*, without any coercion, is the name referring to the second field of the structured value referred to by the value of *draft* and, hence, is not nil.}

5.2.2.1. Syntax

a) **boolean NEST identity relation**{5A} :
**where soft balances SORT1 and SORT2**{32f},
**SORT1 reference to MODE NEST TERTIARY1**{5B},
**identity relator**{b},
**SORT2 reference to MODE NEST TERTIARY2**{5B}.
b) **identity relator**{a} : **is**{94f} **token** ; **is not**{94f} **token.**

{Examples:

a) *next* ***of*** *draft* :=: ***ref book (nil)***
b) :=: • :≠: }

{Observe that *a1* [*i*] :=: *a1* [*j*] is not produced by this syntax. The comparison, by an **identity-relation**, of transient names (2.1.3.6.c) is thus prevented.}

5.2.2.2. Semantics

The yield W of an **identity-relation** I is determined as follows:
• let N1 and N2 be the {collateral} yields of the **TERTIARY**s of I;
Case A: The **token** of the **identity-relator** of I is an **is-token**:
• W is true if {the name} N1 is the same as N2, and is false otherwise;
Case B: The **token** of the **identity-relator** of I is an **is-not-token**:
• W is true if N1 is not the same as N2, and is false, otherwise.

5.2.3. 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., ***loc real*** in *xx* := ***loc real*** := *3.14*, or of a **sample-generator**, e.g., [*1* : *n*] ***char*** in [*1* : *n*] ***char*** *u, v;*, 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 on a run-time stack, whereas **heap-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 less efficient, **local-generators** are preferable; this is why only ***loc*** may be omitted from **sample-generators** of **variable-declarations.**}

5.2.3.1. Syntax

**{LEAP :: local ; heap ; primal.}**

a) **reference to MODE NEST LEAP generator{5C} : LEAP{94d,-} token, actual MODE NEST declarer{46a).**

b) **reference to MODINE NEST LEAP sample generator{44e} : LEAP{94d,-} token, actual MODINE NEST declarer{44b,46a} ; where (LEAP) is (local), actual MODINE NEST declarer{44b,46a).**

{Examples:

a) ***loc real*** b) ***loc real • real*** }

{There is no representation for the **primal-symbol** (see 9.4.a).}

5.2.3.2. Semantics

a) The yield W of a **LEAP-generator** or **LEAP-sample-generator** G, in an environ E, is determined as follows:

- W is a newly created name which is made to refer {2.1.3.2.a} to the yield in E of the **actual-declarer** {4.4.2.d, 4.6.2.a} of G;
- W is equal in scope to the environ E1 determined as follows:

  Case A: '**LEAP**' is '**local**':
  - E1 is the "local environ" {b} accessible from E;

  Case B: '**LEAP**' is '**heap**':
  - E1 is {the first environ created during the elaboration of the **particular-program**, which is} such that

    (i) the primal environ {2.2.2.a} is the environ of the environ of the environ of E1 {sic}, and

    (ii) E1 is, or is older than, E;

  Case C: '**LEAP**' is '**primal**':
  - E1 is the primal environ;
- if W is a stowed name {2.1.3.2.b}, then W is endowed with subnames {2.1.3.3.e, 2.1.3.4.g}.

{The only examples of **primal-generators** occur in the **standard-** and **system-preludes** (10.3.1.1.h, 10.3.1.4.b,n,o, 10.4.1.a).

When G is a **reference-to-routine-sample-generator**, the mode of W is of no relevance.}

b) The "local environ" accessible from an environ E is an environ E1 determined as follows:

If E is "nonlocal" {3.2.2.b},
then E1 is the local environ accessible from the environ of E;
otherwise, E1 is E.

{An environ is nonlocal if it has been established according to a **serial-clause** or **enquiry-clause** which contains no constituent **mode-**, **identifier-**, or **operation-declaration**, or according to a **for-part** (3.5.1.b) or a **specification** (3.4.1.j,k).}

5.2.4. Nihils

5.2.4.1. Syntax

a) **strong reference to MODE NEST nihil{5B} : nil{94f} token.**

{Example:

a) ***nil*** }

5.2.4.2. Semantics

The yield of a **nihil** is a nil name.

5.3. Units associated with stowed values

{In Flanders fields the poppies blow
Between the crosses, row on row, ...
In Flanders Fields, John McCrae.}

{The fields of structured values may be obtained by **selections** (5.3.1) and the elements of multiple values by **slices** (5.3.2); the corresponding effects on stowed names are defined also.}

5.3.1. Selections

{A **selection** selects a field from a structured value or (if it is a "multiple selection") it selects a multiple value from a multiple value whose elements are structured values. For example, *re* ***of*** *z* selects the first real field (usually termed the real part) of the yield of *z*. If *z* yields a name, then *re* ***of*** *z* also yields a name, but if *g* yields a complex value, then *re* ***of*** *g* yields a real value, not a name referring to one.}

5.3.1.1. Syntax

A) **REFETY :: REF to ; EMPTY.**
B) **REFLEXETY :: REF to ; REF to flexible ; EMPTY.**
**{REF :: reference ; transient reference.}**

a) **REFETY MODE1 NEST selection{5C} :**
**MODE1 field FIELDS applied field selector with TAG{48d},**
**of {94f} token, weak REFETY structured with FIELDS mode**
**NEST SECONDARY{5C} ;**
**where (MODE1) is (ROWS of MODE2),**
**MODE2 field FIELDS applied field selector with TAG{48d},**
**of{94f} token, weak REFLEXETY ROWS of structured with**
**FIELDS mode NEST SECONDARY{5C},**
**where (REFETY) is derived from (REFLEXETY){b,c,-}.**

b) **WHETHER (transient reference to) is derived from**
**(REF to flexible){a,532a,66a} : WHETHER true.**

c) **WHETHER (REFETY) is derived from (REFETY){a,532a,66a} :**
**WHETHER true.**

{Examples:

a) *re* **of** *z* • *re* **of** *z1* }

{The mode of *re* **of** *z* begins with '**reference to**' because that of *z* does. Example:

***int*** *age* := *7*; ***struct*** *(**bool** sex, **int** age) jill;*
*age* **of** *jill* := *age*;

Note that the **destination** *age* **of** *jill* yields a name because *jill* yields one. After the **identity-declaration**

***struct*** *(**bool** sex, **int** age) jack* = *(**true**, 9)*,

*age* **of** *jack* cannot be assigned to since *jack* is not a **variable**.}

5.3.1.2. Semantics

The yield W of a **selection** S is determined as follows:

- let V be the yield of the **SECONDARY** of S;
- it is required that V {if it is a name} be not nil;
- W is the value selected in {2.1.3.3.a,e, 2.1.3.4.k} or the name generated from {2.1.3.4.l} V by the **field-selector** of S.

{A selection in a name referring to a structured value yields an existing subname (2.1.3.3.e) of that name. The name generated from a name referring to a multiple value, by way of a **selection** with a **ROWS-of-MODE-SECONDARY** (as in *re* **of** *z1*), is a name which may or may not be newly created for the purpose.}

5.3.2. Slices

{**Slices** are obtained by subscripting, e.g., *x1* [*i*], by trimming, e.g., *x1* [*2*: *n*] or by both, e.g., *x2* [*j*: *n*, *j*] or *x2* [, *k*]. Subscripting and trimming may be done only to **PRIMARYs**, e.g., *x1* or *(p | x1 | y1)* but not *re* **of** *z1*. The value of a **slice** may be either one element of the yield of its **PRIMARY** or a subset of the elements; e.g., *x1* [*i*] is a real number from the row of real numbers *x1*, *x2* [*i*, ] is the *i*-th row of the matrix *x2* and *x2* [, *k*] is its *k*-th column.}

5.3.2.1. Syntax

A) **ROWSETY :: ROWS ; EMPTY.**

a) **REFETY MODE1 NEST slice{5D} :**
**weak REFLEXETY ROWS1 of MODE1 NEST PRIMARY{5D},**
**ROWS1 leaving EMPTY NEST indexer{b,c,-} STYLE bracket,**
**where (REFETY) is derived from (REFLEXETY){531b,c,-} ;**
**where (MODE1) is (ROWS2 of MODE2),**
**weak REFLEXETY ROWS1 of MODE2 NEST PRIMARY{5D},**
**ROWS1 leaving ROWS2 NEST indexer{b,d,-} STYLE bracket,**
**where (REFETY) is derived from (REFLEXETY){531b,c,-}.**
**{ROWS :: row ; ROWS row.}**

b) **row ROWS leaving ROWSETY1 ROWSETY2 NEST indexer**{a,b} :
**row leaving ROWSETY1 NEST indexer**{c,d,-}, **and also**{94f} **token,**
**ROWS leaving ROWSETY2 NEST indexer**{b,c,d,-}.

c) **row leaving EMPTY NEST indexer**{a,b} : **NEST subscript**{e}.

d) **row leaving row NEST indexer**{a,b} : **NEST trimmer**{f} ;
**NEST revised lower bound**{g} **option.**

e) **NEST subscript**{c} : **meek integral NEST unit**{32d}.

f) **NEST trimmer**{d} : **NEST lower bound**{46m} **option, up to**{94f} **token,**
**NEST upper bound**{46n} **option,**
**NEST revised lower bound**{g} **option.**

g) **NEST revised lower bound**{d,f} :
**at**{94f} **token, NEST lower bound**{46m}.

h) ***trimscript : NEST subscript**{e} ; **NEST trimmer**{f} ;
**NEST revised lower bound**{g} **option.**

i) ***indexer : ROWS leaving ROWSETY NEST indexer**{b,c,d}.

j) ***boundscript : NEST subscript**{e} ; **NEST lower bound**{46m} ;
**NEST upper bound**{46n} ; **NEST revised lower bound**{g}.

{Examples:

a) *x2* [*i*, *j*] • *x2* [ , *j*]
b) *1* : *2*, *j* (in *x2* [*1* : *2*, *j*]) • *i*, *j* (in *x2* [*i*, *j*])
c) *j* (in *x2* [*1* : *2*, *j*])
d) *1* : *2* • *@0* (in *x1* [*@0*])
e) *j*
f) *1* : *2* *@0*
g) *@0* }

{A **subscript** decreases the number of dimensions by one, but a **trimmer** leaves it unchanged. In rule a, '**ROWS1**' reflects the number of **trimsripts** in the **slice**, and '**ROWS2**' the number of these which are **trimmers** or **revised-lower-bound-options**.

If the value to be sliced is a name, then the yield of the **slice** is also a name. Moreover, if the mode of the former name is '**reference to flexible ROWS1 of MODE**', then that yield is a transient name (see 2.1.3.6.c).}

5.3.2.2. Semantics

a) The yield W of a **slice** S is determined as follows:

- let V and ($I_1$, ... , $I_n$) be the {collateral} yields of the **PRIMARY** of S and of the **indexer** {b} of S;
- it is required that V {if it is a name} be not nil;
- let (($r_1$, $s_1$), ... , ($r_n$, $s_n$)) be the descriptor of V or of the value referred to by V;

For i = 1, ... , n,

Case A: $I_i$ is an integer:

- it is required that $r_i \leq I_i \leq s_i$;

Case B: $I_i$ is some triplet (l, u, l'):

- let L be $r_i$, if l is absent, and be l otherwise;
- let U be $s_i$, if u is absent, and be u otherwise;
- it is required that $r_i \leq L$ and $U \leq s_i$;
- let D be 0 if l' is absent, and be L − l' otherwise; {D is the amount to be subtracted from L in order to get the revised lower bound;}
- $I_i$ is replaced by (L, U, D);

• W is the value selected in {2.1.3.4.a,g,i} or the name generated from {2.1.3.4.j} V by $(I_1, \ldots, I_n)$.

b) The yield of an **indexer** I of a **slice** S is a trim {2.1.3.4.h} or an index {2.1.3.4.a} $(I_1, \ldots, I_n)$ determined as follows:

• the constituent **boundscripts** of S are elaborated collaterally;
For i = 1, ... , n, where n is the number of constituent **trimscripts** of S,
Case A: the i-th **trimscript** is a **subscript**:

- $I_i$ is {the integer which is} the yield of that **subscript**;

Case B: the i-th **trimscript** is a **trimmer** T:

- $I_i$ is the triplet (l, u, l'), where
  - l is the yield of the constituent **lower-bound**, if any, of T, and is absent, otherwise,
  - u is the yield of the constituent **upper-bound**, if any, of T, and is absent, otherwise,
  - l' is the yield of the constituent **revised-lower-bound**, if any, of T, and is 1, otherwise;

Case C: the i-th **trimscript** is a **revised-lower-bound-option** N:

- $I_i$ is the triplet (absent, absent, l'), where
  - l' is the yield of the **revised-lower-bound**, if any, of N, and is absent otherwise.

{Observe that, if $(I_1, \ldots, I_n)$ contains no triplets, it is an index, and selects one element; otherwise, it is a trim, and selects a subset of the elements.}

{A **slice** from a name referring to a multiple value yields an existing subname (2.1.3.4.j) of that name if all the constituent **trimscripts** of that **slice** are **subscripts**. Otherwise, it yields a generated name which may or may not be newly created for the purpose. Hence, the yield of *x1* [*1* : *2*] :=: *x1* [*1* : *2*] is not defined, although *x1* [*1*] :=: *x1* [*1*] must always yield true.}

{The various possible bounds in the yield of a **slice** are illustrated by the following examples, for each of which the descriptor of the value

referred to by the yield is shown:

*[0 : 9, 2 : 11] **int** i3;*
*i3 [1, 3 : 10 @3] ¢ ((3, 10)) ¢;*
*i3 [1, 3 : 10] ¢ ((1, 8)) ¢;*
*i3 [1, 3 : ] ¢ ((1, 9)) ¢;*
*i3 [1, : ] ¢ ((1, 10)) ¢;*
*i3 [1, ] ¢ ((2, 11)) ¢;*
*i3 [ , 2] ¢ ((0, 9)) ¢.}*

## 5.4. Units associated with routines

{Routines are created from **routine-texts** (5.4.1) or from **jumps** (5.4.4), and they may be "called" by **calls** (5.4.3), **formulas** (5.4.2) or by deproceduring (6.3).}

### 5.4.1. Routine texts

{A **routine-text** always has a **formal-declarer**, specifying the mode of the result, and a **routine-token**, viz., a colon. To the right of this colon stands a **unit**, which prescribes the computations to be performed when the routine is called. If there are parameters, then to the left of the **formal-declarer** stands a **declarative** containing the various **formal-parameters** required.
Examples:

***void**: print (x);*
*(**ref real** a, **real** b) **bool**: (a < b | a := b; **true** | **false**).}*

#### 5.4.1.1. Syntax

a) **procedure yielding MOID NEST1 routine text{44d,5A} :**
**formal MOID NEST1 declarer{46b}, routine{94f} token,**
**strong MOID NEST1 unit{32d}.**

b) **procedure with PARAMETERS yielding**
**MOID NEST1 routine text{44d,5A} :**
**NEST1 new DECS2 declarative defining**
**new DECS2{e} brief pack,**
**where DECS2 like PARAMETERS{c,d,-},**
**formal MOID NEST1 declarer{46b}, routine{94f} token,**
**strong MOID NEST1 new DECS2 unit{32d}.**

c) **WHETHER DECS DEC like PARAMETERS PARAMETER{b,c} :**
**WHETHER DECS like PARAMETERS{c,d,-}**
**and DEC like PARAMETER{d,-}.**
**{PARAMETER :: MODE parameter.}**

d) **WHETHER MODE TAG like MODE parameter{b,c} :**
**WHETHER true.**

e) **NEST2 declarative defining new DECS2{b,e,34j} :**
**formal MODE NEST2 declarer{46b},**
**NEST2 MODE parameter joined definition of DECS2{41b,c} ;**

**where (DECS2) is (DECS3 DECS4),**
**formal MODE NEST2 declarer{46b},**
**NEST2 MODE parameter joined definition of DECS3{41b,c},**
**and also{94f} token, NEST2 declarative defining new DECS4{e}.**

f) **NEST2 MODE parameter definition of MODE TAG2{41c} :**
**MODE NEST2 defining identifier with TAG2{48a}.**

g) ***formal MODE parameter :**
**NEST MODE parameter definition of MODE TAG{f}.**

{Examples:

a) ***real***: *random* × *10* b) *(**bool** a, b)**bool**: (a | b | **false**)*
e) ***bool** a, b • **bool** a, **bool** b* f) *a* }

5.4.1.2. Semantics

The yield of a **routine-text** T, in an environ E, is the routine composed of
(i) T, and
(ii) the environ necessary for {7.2.2.c} T in E.

5.4.2. Formulas

{**Formulas** are either dyadic or monadic: e.g., $x + i$ or ***abs*** $x$. The order of elaboration of a **formula** is determined by the priority of its **operators**; monadic **formulas** are elaborated first and then the dyadic ones from the highest to the lowest priority.}

5.4.2.1. Syntax

A) **DYADIC :: priority PRIO.**
B) **MONADIC :: priority iii iii iii i.**
C) **ADIC :: DYADIC ; MONADIC.**
D) **TALLETY :: TALLY ; EMPTY.**

a) **MOID NEST DYADIC formula{c,5B} :**
**MODE1 NEST DYADIC TALLETY operand{c,-},**
**procedure with MODE1 parameter MODE2 parameter**
**yielding MOID NEST applied operator with TAD{48b},**
**where DYADIC TAD identified in NEST{72a},**
**MODE2 NEST DYADIC TALLY operand{c,-}.**

b) **MOID NEST MONADIC formula{c,5B} :**
**procedure with MODE parameter yielding MOID**
**NEST applied operator with TAM {48b},**
**MODE NEST MONADIC operand{c}.**

c) **MODE NEST ADIC operand{a,b} :**
**firm MODE NEST ADIC formula{a,b} coercee{61b} ;**
**where (ADIC) is (MONADIC), firm MODE NEST SECONDARY{5C}.**

d) ***MOID formula : MOID NEST ADIC formula**{a,b}.
e) ***DUO dyadic operator with TAD :**
**DUO NEST DEFIED operator with TAD**{48a,b}.
f) ***MONO monadic operator with TAM :**
**MONO NEST DEFIED operator with TAM**{48a,b}.
g) ***MODE operand : MODE NEST ADIC operand**{c}.

{Examples:

a) $-x + 1$ b) $-x$
c) $-x \bullet 1$ }

5.4.2.2. Semantics

The yield W of a **formula** F, in an environ E, is determined as follows:
- let R be the routine yielded in E by the **operator** of F;
- let $V_1$, ... , $V_n$ {n is 1 or 2} be the {collateral} yields of the **operands** of F,

in an environ E1 established {locally, see 3.2.2.b} around E;
- W is the yield of the calling {5.4.3.2.b} of R in E1, with $V_1$, ... , $V_n$;
- it is required that W be not newer in scope than E.

{Observe that $a \uparrow b$ is not precisely the same as $a^b$ in the usual notation; indeed, the value of *(- 1 ↑ 2 + 4 = 5)* and that of *(4 - 1 ↑ 2 = 3)* both are true, since the first **minus-symbol** is a **monadic-operator**, whereas the second is a **dyadic-operator**.}

5.4.3. Calls

{**Calls** are used to command the elaboration of routines parametrized with **actual-parameters**.
Examples:
*sin (x)* • *(p | sin | cos) (x).*}

5.4.3.1. Syntax

a) **MOID NEST call**{5D} **: meek procedure with PARAMETERS yielding**
**MOID NEST PRIMARY**{5D},
**actual NEST PARAMETERS**{b,c} **brief pack.**
b) **actual NEST PARAMETERS PARAMETER**{a,b} **:**
**actual NEST PARAMETERS**{b,c}**, and also**{94f} **token,**
**actual NEST PARAMETER**{c}.
c) **actual NEST MODE parameter**{a,b} **: strong MODE NEST unit**{32d}.

{Examples:

a) *put (stand out, x)* (see 10.3.3.1.a)
b) *stand out, x* c) *x* }

5.4.3.2. Semantics

a) The yield W of a **call** C, in an environ E, is determined as follows:

- let R {a routine} and $V_1$, ... , $V_n$ be the {collateral} yields of the **PRIMARY** of C, in E, and of the constituent **actual-parameters** of C, in an environ E1 established {locally, see 3.2.2.b} around E;
- W is the yield of the calling {b} of R in E1 with $V_1$, ... , $V_n$;
- it is required that W be not newer in scope than E.

b) The yield W of the "calling" of a routine R in an environ E1, possibly with {parameter} values $V_1$, ... , $V_n$, is determined as follows:

- let E2 be the environ established {3.2.2.b} upon E1, around the environ of R, according to the **declarative** of the **declarative-pack**, if any, of the **routine-text** of R, with the values $V_1$, ... , $V_n$, if any;
- W is the yield in E2 of the **unit** of the **routine-text** of R.

{Consider the following **serial-clause**:

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

In that context, the last **call** has the same effect as the following **cast**:

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

The transmission of **actual-parameters** is thus similar to the elaboration of **identity-declarations** (4.4.2.a); see also establishment (3.2.2.b) and ascription (4.8.2.a).}

5.4.4. Jumps

{A **jump** may terminate the elaboration of a **series** and cause some other labelled **series** to be elaborated in its place.

Examples:

```
y := if x ≥ 0 then sqrt (x) else goto princeton fi •
goto st pierre de chartreuse.
```

Alternatively, if the context expects the mode '**procedure yielding MOID**', then a routine whose **unit** is that **jump** is yielded instead, as in

```
proc void m := goto north berwick.
```

}

5.4.4.1. Syntax

a) **strong MOID NEST jump**{5A} : **go to**{b} **option,**
**label NEST applied identifier with TAG**{48b}.

b) **go to**{a} : **STYLE go to**{94f,-} **token** ;
**STYLE go**{94f,-} **token, STYLE to symbol**{94g,-}.

{Examples:

a) ***goto*** *kootwijk* • ***go to*** *warsaw* • *zandvoort*
b) ***goto*** • ***go to*** }

5.4.4.2. Semantics

A **MOID-NEST-jump** J, in an environ E, is elaborated as follows:
• let the scene yielded in E by the **label-identifier** of J be composed of a **series** S2 and an environ E1;
Case A: '**MOID**' is not any '**procedure yielding MOID1**':
• let S1 be the **series** of the smallest {1.1.3.2.g} **serial-clause** containing S2;
• the elaboration of S1 in E1, or of any series in E1 elaborated in its place, is terminated {2.1.4.3.e};
• S2 in E1 is elaborated "in place of" S1 in E1;
Case B: '**MOID**' is some '**procedure yielding MOID1**':
• J in E {is completed and} yields the routine composed of
(i) a new **MOID-NEST-routine-text** whose **unit** is akin {1.1.3.2.k} to J,
(ii) E1.

5.5. Units associated with values of any mode

5.5.1. Casts

{**Casts** may be used to provide a strong position. For example, ***ref real*** *(xx)* in ***ref real*** *(xx)* := *1*, ***ref book*** *(nil)* in *next* ***of*** *draft* :=: ***ref book*** *(nil)* and ***string*** *(p | c | r)* in *s* +:= ***string*** *(p | c | r).*}

5.5.1.1. Syntax

a) **MOID NEST cast**{5D} : **formal MOID NEST declarer**{46b},
**strong MOID NEST ENCLOSED clause**{31a,33a,c,d,e,34a,35a,-}.

{Example:

a) ***ref book*** *(nil)* }

{The yield of a **cast** is that of its **ENCLOSED-clause**, by way of pre-elaboration (2.1.4.1.c).}

5.5.2. Skips

5.5.2.1. Syntax

a) **strong MOID NEST skip**{5A} : **skip**{94f} **token.**

5.5.2.2. Semantics

The yield of a **skip** is some {undefined} value equal in scope to the primal environ.

{The mode of the yield of a **MOID-skip** is '**MOID**'. A **void-skip** serves as a dummy statement and may be used, for example, after a **label** which marks the end of a **serial-clause**.}

# PART III

## Context Dependence

{This Part deals with those rules which do not alter the underlying syntactical structure:
- the transformations of modes implicitly defined by the context, with their accompanying actions;
- the syntax needed for the equivalence of modes and for the safe application of the properties kept in the nests.}

### 6. Coercion

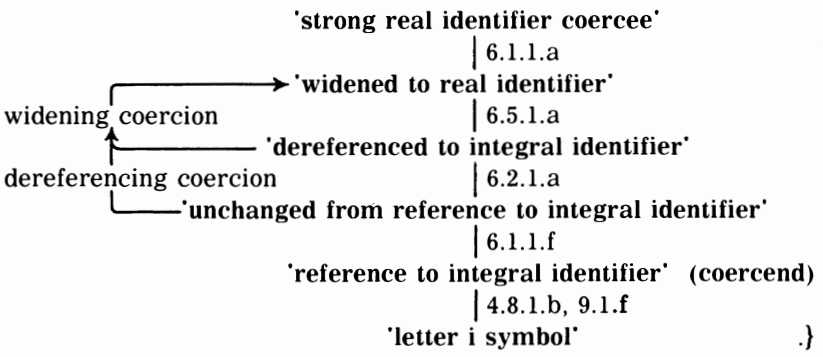
{The coercions produce a **coercend** from a **coercee** according to three criteria: the a priori mode of the **coercend** before the application of any coercion, the a posteriori mode of the **coercee** required after those coercions, and the syntactic position or "sort" of the **coercee**. Coercions may be cascaded.

There are six possible coercions, termed "deproceduring", "dereferencing", "uniting", "widening", "rowing" and "voiding". Each coercion, except "uniting", prescribes a corresponding dynamic effect on the associated values. Hence, a number of primitive actions can be programmed implicitly by coercions.}

6.1. Coercees

{A **coercee** is a construct whose production tree may begin a sequence of coercions ending in a **coercend**. The order of (completion of) the elaboration of the coercions is therefore from the **coercend** to the **coercee** (hence the choice of these paranotions). For example, *i* in ***real*** *(i)* is a **coercee** whose production tree involves '**widened to**' and '**dereferenced to**', in that order, in passing from the **coercee** to the **coercend**. Note that the dereferencing must be completed before the widening takes place.

The relevant production tree (with elision of **'NEST'**, **'applied'** and **'with TAG'**, and with invisible subtrees omitted) is:




6.1.1. Syntax

A) **STRONG**{a,66a} :: **FIRM**{B} ; **widened to**{65a,b,c,d} ; **rowed to**{66a} ; **voided to**{67a,b}.
B) **FIRM**{A,b} :: **MEEK**{C} ; **united to**{64a}.
C) **MEEK**{B,c,d,62a,63a,64a,65a,b,c,d} :: **unchanged from**{f} ; **dereferenced to**{62a} ; **deprocedured to**{63a}.
D) **SOFT**{e,63b} :: **unchanged from**{f} ; **softly deprocedured to**{63b}.
E) **FORM :: MORF ; COMORF.**
F) **MORF :: NEST selection ; NEST slice ; NEST routine text ; NEST ADIC formula ; NEST call ; NEST applied identifier with TAG.**
G) **COMORF :: NEST assignation ; NEST identity relation ; NEST LEAP generator ; NEST cast ; NEST denoter ; NEST format text.**

a) **strong MOID FORM coercee**{5A,B,C,D,A341i} :
**where (FORM) is (MORF), STRONG**{A} **MOID MORF ;**
**where (FORM) is (COMORF), STRONG**{A} **MOID COMORF,**
**unless (STRONG MOID) is (deprocedured to void).**
b) **firm MODE FORM coercee**{5A,B,C,D,542c} : **FIRM**{B} **MODE FORM.**
c) **meek MOID FORM coercee**{5A,B,C,D} : **MEEK**{C} **MOID FORM.**
d) **weak REFETY STOWED FORM coercee**{5A,B,C,D} :
**MEEK**{C} **REFETY STOWED FORM,**
**unless (MEEK) is (dereferenced to)**
**and (REFETY) is (EMPTY).**
e) **soft MODE FORM coercee**{5A,B,C,D} : **SOFT**{D} **MODE FORM.**
f) **unchanged from MOID FORM**{C,D,67a,b} : **MOID FORM.**

g) ***SORT MOID coercee : SORT MOID FORM coercee**{a,b,c,d,e}.
h) ***MOID coercend : MOID FORM.**

{Examples:

a) *3.14* (in *x* := *3.14*)
b) *3.14* (in *x* + *3.14*)
c) *sin* (in *sin (x)*)
d) *x1* (in *x1* [*2*] := *3.14*)
e) *x* (in *x* := *3.14*) }

{For **'MOID FORM'** (rule f), see the cross-references inserted in sections 5.1.A,B,C,D before "**coercee**". Note, however, that a **'MOID FORM'** may be a blind alley. Blind alleys within this chapter are not indicated.}

{There are five sorts of syntactic position. They are:
- "strong" positions, i.e., **actual-parameters**, e.g., *x* in *sin (x)*, **sources**, e.g., *x* in *y* := *x*, the **ENCLOSED-clause** of a **cast**, e.g., ***(nil)*** in ***ref book (nil)***, and **statements**, e.g., *y* := *x* in *(y* := *x; x* := *0)*;
- "firm" positions, i.e., **operands**, e.g., *x* in *x* + *y*;
- "meek" positions, i.e., **enquiry-clauses**, e.g., *x* > *0* in *(x* > *0* | *x* | *0)*, **boundscripts**, e.g., *i* in *x1* [*i*], and the **PRIMARY** of a **call**, e.g., *sin* in *sin (x)*;
- "weak" positions, i.e., the **SECONDARY** of a **selection** and the **PRIMARY** of a **slice**, e.g., *x1* in *x1* [*i*];
- "soft" positions, i.e., **destinations**, e.g., *x* in *x* := *y* and one of the **TERTIARYs** of an **identity-relation**, e.g., *x* in *xx* :=: *x*.

Strong positions also arise in balancing (3.2.1.e).

In strong positions, all six coercions may occur; in firm positions, rowing, widening and voiding are forbidden; in meek and weak positions, uniting is forbidden also, and in soft positions only deproceduring is allowed. However, a **dereferenced-to-STOWED-FORM** may not be directly descended from a **weak-STOWED-FORM-coercee** (rule d) for, otherwise, *x* := *x1* [*i*] would be syntactically ambiguous (although, in this case, not semantically). Also, a **deprocedured-to-void-COMORF** may not be directly descended from a **strong-void-COMORF-coercee** (rule a) for, otherwise,

*(**proc void** engelfriet; **proc void** rijpens = **skip**; engelfriet := rijpens; **skip**)*

would be ambiguous.}

## 6.2. Dereferencing

{Dereferencing serves to obtain the value referred to by a name, as in *x* := *y*, where *y* yields a name referring to a real number and it is this number which is assigned to the name yielded by *x*. The a priori mode of *y*, regarded as a **coercend**, is **'reference to real'** and its a posteriori mode, when *y* is regarded as a **coercee**, is **'real'**.}

### 6.2.1. Syntax

a) **dereferenced to{61C} MODE1 FORM :**
**MEEK{61C} REF to MODE2 FORM,**
**where MODE2 deflexes to MODE1{47a,b,c,-}.**

{Example:

a) *x* (in ***real*** *(x)*)}

6.2.2. Semantics

The yield W of a **dereferenced-to-MODE-FORM** F is determined as follows:

- let {the name} N be the yield of the **MEEK-FORM** of F;
- it is required that N be not nil;
- W is the value referred to by N.

6.3. Deproceduring

{Deproceduring is used when a routine without parameters is to be called. E.g., in *x* := *random*, the routine yielded by *random* is called and the real number yielded is assigned: the a posteriori mode of *random* is **'real'**. Syntactically, an initial **'procedure yielding'** is removed from the a priori mode.}

6.3.1. Syntax

a) **deprocedured to**{61C,67a} **MOID FORM :**
**MEEK**{61C} **procedure yielding MOID FORM.**

b) **softly deprocedured to**{61D} **MODE FORM :**
**SOFT**{61D} **procedure yielding MODE FORM.**

{Examples:

a) *random* (in ***real*** *(random)*)
b) *x or y* (in *x or y* := *3.14*, see 1.1.2) }

6.3.2. Semantics

The yield W of a **deprocedured-to-MOID-FORM** or **softly-deprocedured-to-MOID-FORM** F, in an environ E, is determined as follows:

- let {the routine} R be the yield in E of the direct descendent of F;
- W is the yield of the calling {5.4.3.2.b} of R in E;
- it is required that W be not newer in scope than E.

6.4. Uniting

{Uniting does not change the mode of the run-time value yielded by a construct, but simply gives more freedom to it. That value must be acceptable to not just that one mode, but rather to the whole of a given set of modes. However, after uniting, that value may be subject to a primitive action only after being dynamically tested in a **conformity-clause** (3.4.1.q); indeed, no primitive action can be programmed with a construct of a **'UNITED'** mode (except to assign it to a **UNITED-variable**, of course).

Example:

***union (bool, char)*** *t, u;*
*t* := "a"; *t* := ***true***; *u* := *t*. }

6.4.1. Syntax

a) **united to{61B} UNITED FORM : MEEK{61C} MOID FORM,**
**where MOID unites to UNITED{b}.**

b) **WHETHER MOID1 unites to MOID2{a,34i,71m} :**
**where MOID1 equivalent MOID2{73a}, WHETHER false ;**
**unless MOID1 equivalent MOID2{73a},**
**WHETHER safe MOODS1 subset of safe MOODS2{73l,m,n},**
**where (MOODS1) is (MOID1)**
**or (union of MOODS1 mode) is (MOID1),**
**where (MOODS2) is (MOID2)**
**or (union of MOODS2 mode) is (MOID2).**

{Examples:

a) *x* (in *uir* := *x*) •
*u* (in ***union (char, int, void)*** *(u)*, in a reach containing
***union (int, void)*** *u* := ***empty***)}

6.5. Widening

{Widening transforms integers to real numbers, real numbers to complex numbers (in both cases, with the same size), a value of mode **'BITS'** to an unpacked vector of truth values, or a value of mode **'BYTES'** to an unpacked vector of characters.

For example, in *z* := *1*, the yield of *1* is widened to the real number 1.0 and then to the complex number (1.0, 0.0); syntactically, the a priori mode specified by ***int*** is changed to that specified by ***real*** and then to that specified by ***compl***.}

6.5.1. Syntax

A) **BITS :: structured with**
**row of boolean field SITHETY letter aleph mode.**

B) **BYTES :: structured with**
**row of character field SITHETY letter aleph mode.**

C) **SITHETY :: LENGTH LENGTHETY ; SHORTH SHORTHETY ;**
**EMPTY.**

D) **LENGTH :: letter l letter o letter n letter g.**

E) **SHORTH :: letter s letter h letter o letter r letter t.**

F) **LENGTHETY :: LENGTH LENGTHETY ; EMPTY.**

G) **SHORTHETY :: SHORTH SHORTHETY ; EMPTY.**

a) **widened to{b,61A} SIZETY real FORM :**
**MEEK{61C} SIZETY integral FORM.**
**{SIZETY :: long LONGSETY ; short SHORTSETY ; EMPTY.}**

b) **widened to**{61A} **structured with SIZETY real field letter r letter e SIZETY real field letter i letter m mode FORM :**
**MEEK**{61C} **SIZETY real FORM ;**
**widened to**{a} **SIZETY real FORM.**

c) **widened to**{61A} **row of boolean FORM : MEEK**{61C} **BITS FORM.**

d) **widened to**{61A} **row of character FORM : MEEK**{61C} **BYTES FORM.**

{Examples:

a) *1* (in *x* := *1*)
b) *1.0* (in *z* := *1.0*) • *1* (in *z* := *1*)
c) *2r101* (in [ ] ***bool*** *(2r101)*)
d) *r* (in [ ] ***char*** *(r)*, see 1.1.2)}

6.5.2. Semantics

The yield W of a **widened-to-MODE-FORM** F is determined as follows:
• let V be the yield of the direct descendent of F;
Case A: '**MODE**' is some '**SIZETY real**':
• W is the real number widenable from {2.1.3.1.e} V;
Case B: '**MODE**' is some '**structured with SIZETY real letter r letter e SIZETY real letter i letter m mode**':
• W is {the complex number which is} a structured value whose fields are respectively V and the real number 0 of the same size {2.1.3.1.b} as V;
Case C: '**MODE**' is '**row of boolean**' or '**row of character**':
• W is the {only} field of V.

6.6. Rowing

{Rowing permits the building of a multiple value from a single element. If the latter is a name then the result of rowing may also be a name referring to that multiple value.
Example:
[*1* : *1*] ***real*** *b1* := *4.13* }

6.6.1. Syntax

a) **rowed to**{61A} **REFETY ROWS1 of MODE FORM :**
**where (ROWS1) is (row),**
**STRONG**{61A} **REFLEXETY MODE FORM,**
**where (REFETY) is derived from (REFLEXETY)**{531b,c,-} ;
**where (ROWS1) is (row ROWS2),**
**STRONG**{61A} **REFLEXETY ROWS2 of MODE FORM,**
**where (REFETY) is derived from (REFLEXETY)**{531b,c,-}.

{Examples:

a) *4.13* (in [*1* : *1*] ***real*** *b1* := *4.13*) •
*x1* (in [*1* : *1*, *1* : *n*] ***real*** *b2* := *x1*)}

6.6.2. Semantics

a) The yield W of a **rowed-to-REFETY-ROWS1-of-MODE-FORM** F is determined as follows:
• let V be the yield of the **STRONG-FORM** of F;
Case A: **'REFETY' is 'EMPTY'**:
• W is the multiple value "built" {b} from V for **'ROWS1'**;
Case B: **'REFETY' is 'REF to'**:
If V is nil,
then W is a nil name;
otherwise, W is the name "built" {c} from V for **'ROWS1'**.

b) The multiple value W "built" from a value V, for some **'ROWS1'**; is determined as follows:
Case A: **'ROWS1' is 'row'**:
• W is composed of
(i) a descriptor ((1, 1)),
(ii) {one element} V;
Case B: **'ROWS1' is some 'row ROWS2'**:
• let the descriptor of V be $((l_1, u_1), \ldots, (l_n, u_n))$;
• W is composed of
(i) a descriptor $((1, 1), (l_1, u_1), \ldots, (l_n, u_n))$,
(ii) the elements of V;
• the element selected by an index $(i_1, \ldots, i_n)$ in V is that selected by $(1, i_1, \ldots, i_n)$ in W.

c) The name N1 "built" from a name N, for some **'ROWS1'**, is determined as follows:
• N1 is a name {not necessarily newly created}, equal in scope to N and referring to the multiple value built {b}, for **'ROWS1'**, from the value referred to by N;
Case A: **'ROWS1' is 'row'**:
• the {only} subname of N1 is N;
Case B: **'ROWS1' is some 'row ROWS2'**:
• the subname of N1 selected by $(1, i_1, \ldots, i_n)$ is the subname of N selected by $(i_1, \ldots, i_n)$.

6.7. Voiding

{Voiding is used to discard the yield of some **unit** whose primary purpose is to cause its side-effects; the a posteriori mode is then simply **'void'**. For example, in *x := 1; y := 1;*, the **assignation** *y := 1* is voided, and in ***proc*** *t* = ***int***: ***entier***(*random* × *100*); *t;*, the **applied-identifier** *t* is voided after a deproceduring, which prescribes the calling of a routine.

**Assignations** and other **COMORFs** are voided without any deproceduring so that, in ***proc void*** *p; .p* := *finish*, the **assignation** *p* := *finish* does not prescribe an unexpected calling of the routine *finish*.}

6.7.1. Syntax

A) **NONPROC :: PLAIN ; STOWED ; REF to NONPROC ;**
**procedure with PARAMETERS yielding MOID ; UNITED.**

a) **voided to**{61A} **void MORF : deprocedured to**{63a} **NONPROC MORF ;**
**unchanged from**{61f} **NONPROC MORF.**

b) **voided to**{61A} **void COMORF :**
**unchanged from**{61f} **MODE COMORF.**

{Examples:

a) *random* (in ***skip***; *random;*) •
*next random (last random)*
(in ***skip***; *next random (last random);*)

b) ***proc void*** *(pp)* (in ***proc proc void*** *pp* = ***proc void*** : *(print (1);*
***void*** : *print (2)); **proc void** (pp);*)}

6.7.2. Semantics

The elaboration of a **voided-to-void-FORM** consists of that of its direct descendent, and yields empty.

## 7. Modes and nests

{The identification of a property in a nest is the static counterpart of the dynamic determination (4.8.2.b) of a value in an environ: the search is conducted from the newest (youngest) level towards the previous (older) ones.

Modes are composed from the primitive modes, such as **'boolean'**, with the aid of **'HEAD'**s, such as **'structured with'**, and they may be recursive. Recursive modes spelled in different ways may nevertheless be equivalent. The syntax tests the equivalence of such modes by proving that it is impossible to find any discrepancy between their respective structures or component modes.

A number of unsafe uses of properties are prevented. An **identifier** or **mode-indication** is not declared more than once in each reach. The modes of the **operands** of a **formula** do not determine more than one operation. Recursions in modes do not cause the creation of dynamic objects of unlimited size and do not allow ambiguous coercions.}

7.1. Independence of properties

{The following syntax determines whether two properties (i.e., two **'PROP'**s), such as those corresponding to ***real*** *x* and ***int*** *x*, may or may not be enveloped by the same **'LAYER'**.}

7.1.1. Syntax

A) **PREF :: procedure yielding ; REF to.**
B) **NONPREF :: PLAIN ; STOWED ;**
**procedure with PARAMETERS yielding MOID ; UNITED ; void.**
C) ***PREFSETY :: PREF PREFSETY ; EMPTY.**

**{PROP :: DEC ; LAB ; FIELD.**
**QUALITY :: MODE ; MOID TALLY ; DYADIC ; label ; MODE field.**
**TAX :: TAG ; TAB ; TAD ; TAM.**
**TAO :: TAD ; TAM.}**

a) **WHETHER PROP1 independent PROPS2 PROP2{a,48a,c,72a} :**
**WHETHER PROP1 independent PROPS2{a,c}**
**and PROP1 independent PROP2{c}.**
b) **WHETHER PROP independent EMPTY{48a,c,72a} : WHETHER true.**
c) **WHETHER QUALITY1 TAX1**
**independent QUALITY2 TAX2{a,48a,c,72a} :**
**unless (TAX1) is (TAX2), WHETHER true ;**
**where (TAX1) is (TAX2) and (TAX1) is (TAO),**
**WHETHER QUALITY1 independent QUALITY2{d}.**
d) **WHETHER QUALITY1 independent QUALITY2{c} :**
**where QUALITY1 related QUALITY2{e,f,g,h,i,j,-},**
**WHETHER false ;**
**unless QUALITY1 related QUALITY2{e,f,g,h,i,j,-},**
**WHETHER true.**

e) **WHETHER MONO related DUO{d} : WHETHER false.**
f) **WHETHER DUO related MONO{d} : WHETHER false.**
g) **WHETHER PRAM related DYADIC{d} : WHETHER false.**
h) **WHETHER DYADIC related PRAM{d} : WHETHER false.**
i) **WHETHER procedure with MODE1 parameter MODE2 parameter**
**yielding MOID1 related**
**procedure with MODE3 parameter MODE4 parameter**
**yielding MOID2{d} :**
**WHETHER MODE1 firmly related MODE3{k}**
**and MODE2 firmly related MODE4{k}.**
j) **WHETHER procedure with MODE1 parameter yielding MOID1**
**related procedure with MODE2 parameter yielding**
**MOID2{d} : WHETHER MODE1 firmly related MODE2{k}.**

k) **WHETHER MOID1 firmly related MOID2{i,j} :**
**WHETHER MOODS1 is firm MOID2{l,m}**
**or MOODS2 is firm MOID1{l,m},**
**where (MOODS1) is (MOID1)**
**or (union of MOODS1 mode) is (MOID1),**
**where (MOODS2) is (MOID2)**
**or (union of MOODS2 mode) is (MOID2).**

l) **WHETHER MOODS MOOD is firm MOID{k,l} :**
**WHETHER MOODS is firm MOID{l,m}**
**or MOOD is firm MOID{m}.**

m) **WHETHER MOID1 is firm MOID2{k,l,n,47f} :**
**WHETHER MOID1 equivalent MOID2{73a}**
**or MOID1 unites to MOID2{64b}**
**or MOID1 deprefs to firm MOID2{n}.**

n) **WHETHER MOID1 deprefs to firm MOID2{m} :**
**where (MOID1) is (PREF MOID3),**
**WHETHER MOID5 is firm MOID2{m},**
**where MOID3 deflexes to MOID5{47a,b,c} ;**
**where (MOID1) is (NONPREF), WHETHER false.**

{To prevent the ambiguous application of **indicators**, as in ***real*** *x*, ***int*** *x*; *x* := *0*, certain restrictions are imposed on **defining-indicators** contained in a given reach. These are enforced by the syntactic test for "independence" of properties enveloped by a given **'LAYER'** (rules a, b, c). A sufficient condition, not satisfied in the example above, for the independence of a pair of properties, each being some **'QUALITY TAX'**, is that the **'TAX'**s differ (rule c). For **'TAX'**s which are not some **'TAO'**, this condition is also necessary, so that even ***real*** *x*, ***int*** *x*; ***skip*** is not a **serial-clause**.

For two properties **'QUALITY1 TAO'** and **'QUALITY2 TAO'** the test for independence is more complicated, as is exemplified by the **serial-clause**

***op*** + = ***(int*** *i)* ***bool*** : ***true***, ***op*** + = ***(int*** *i, j)* ***int*** : *1*, ***op*** + = ***(int*** *i,* ***bool*** *j)* ***int*** : *2*,
***prio*** + = *6*;
*0* + + *0* ¢ = *2* ¢ .

Ambiguities would be present in

***prio*** + = *6*, + = *7*; *1* + *2* × *3* ¢ *7 or 9*? ¢ ,

in

***op z*** = ***(int*** *i)* ***int*** : *1*, ***mode z*** = ***int***;
***z*** *i* ¢ *formula or declaration*? ¢ ; ***skip*** ,

and in

***op*** ? = ***(union (ref real, char)*** *a)* ***int*** : *1*, ***op*** ? = ***(real*** *a)* ***int*** : *2*;
? ***loc real*** ¢ *1 or 2*? ¢ .

In such cases a test is made that the two **'QUALITY'**s are independent (rules c, d). A **'MOID TALLY'** is never independent of any **'QUALITY'** (rule d). A **'MONO'** is always independent of a **'DUO'** (rules d, e, f) and both are independent of a **'DYADIC'** (i.e., of a **'priority PRIO'**) (rules d, g, h). In the case of two **'PRAM'**s which are both **'MONO'** or both **'DUO'**, ambiguities could arise if the corresponding parameter modes were "firmly related", i.e., if some (pair of) operand mode(s) could be firmly coerced to the (pair of) parameter mode(s) of either **'PRAM'** (rules i, j). In the example with the two definitions of ?, the two **'PRAM'**s are related since the modes specified by ***union (ref real, char)*** and by ***real*** are firmly related, the mode specified by ***ref real*** being firmly coercible to either one.

It may be shown that two modes are firmly related if one of them, or some component **'MOOD'** of one of them, may be firmly coerced to the

other (rules k, l), which requires a sequence of zero or more meek coercions followed by at most one uniting (6.4.1.a). The possibility or otherwise of such a sequence of coercions between two modes is determined by the predicate **'is firm'** (rules m, n).

A **'PROP1'** also renders inaccessible a **'PROP2'** in an outer **'LAYER'** if that **'PROP2'** is not independent of **'PROP1'**; e.g.,

```
begin int x;
    begin real x; ¢ here the 'PROP1' is 'reference to real letter x' ¢
        skip
    end
end
```

and likewise

```
begin op ? = (int i) int : 1, int k := 2;
    begin op ? = (ref int i) int : 3;
        ? k ¢ delivers 3, but ? 4 could not occur here because its
            operator is inaccessible ¢
    end
end .)
```

7.2. Identification in nests

{This section ensures that for each **applied-indicator** there is a corresponding property in some suitable **'LAYER'** of the nest.}

7.2.1. Syntax

```
{PROPSETY :: PROPS ; EMPTY.
PROPS :: PROP ; PROPS PROP.
PROP :: DEC ; LAB ; FIELD.
QUALITY :: MODE ; MOID TALLY ; DYADIC ; label ; MODE field.
TAX :: TAG ; TAB ; TAD ; TAM.}

a)  WHETHER PROP identified in NEST new PROPSETY{a,48b,542a} :
      where PROP resides in PROPSETY{b,c,-}, WHETHER true ;
      where PROP independent PROPSETY{71a,b,c},
          WHETHER PROP identified in NEST{a,-}.

b)  WHETHER PROP1 resides in PROPS2 PROP2{a,b,48d} :
      WHETHER PROP1 resides in PROP2{c,-}
          or PROP1 resides in PROPS2{b,c,-}.

c)  WHETHER QUALITY1 TAX resides in QUALITY2 TAX{a,b,48d} :
      where (QUALITY1) is (label) or (QUALITY1) is (DYADIC)
          or (QUALITY1) is (MODE field),
          WHETHER (QUALITY1) is (QUALITY2) ;
      where (QUALITY1) is (MOID1 TALLETY)
          and (QUALITY2) is (MOID2 TALLETY),
          WHETHER MOID1 equivalent MOID2{73a}.
```

{A nest, except the primal one (which is just **'new'**), is some **'NEST LAYER'** (i.e., some **'NEST new PROPSETY'**). A **'PROP'** is identified by first looking for it in that **'LAYER'** (rule a). If the **'PROP'** is some **'label TAX'** or **'DYADIC TAX'**, then a simple match of the **'PROP'**s is a sufficient test (rule c). If the **'PROP'** is some **'MOID TALLETY TAX'**, then the mode equivalencing mechanism must be invoked (rule c). If it is not found in the **'LAYER'**, then the search continues with the **'NEST'** (without that **'LAYER'**), provided that it is independent of all **'PROP'**s in that **'LAYER'**; otherwise the search is abandoned (rule a). Note that rules b and c do double duty in that they are also used to check the validity of **applied-field-selectors** (4.8.1.d).}

7.2.2. Semantics

a) If some **NEST-range** R {3.0.1.f} contains an **applied-indicator** I {4.8.1.b} of which there is a descendent **where-PROP-identified-in-NEST-LAYER**, but no descendent **where-PROP-identified-in-NEST**, then R is the "defining **range**" of that I. {Note that **'NEST'** is always the nest in force just outside the **range**.}

b) A **QUALITY-applied-indicator-with-TAX** I whose defining **NEST-range** {a} is R "identifies" the **QUALITY-NEST-LAYER-defining-indicator-with-TAX** contained in R.

{For example, in

*(¢1¢* ***real*** *i = 2.0; (¢2¢* ***int*** *i = 1; (¢3¢* ***real*** *x; print (i) ¢3¢) ¢2¢) ¢1¢)*

there are three **ranges**. The **applied-identifier** *i* in *print (i)* is forced, by the syntax, to be an **integral-NEST-new-real-letter-i-new-integral-letter-i-new-reference-to-real-letter-x-applied-identifier-with-letter-i** (4.8.1.b). Its defining **range** is the **NEST-new-real-letter-i-serial-clause-defining-new-integral-letter-i** (3.2.1.a) numbered *¢2¢*, it identifies the **defining-identifier** *i* contained in ***int*** *i* (not the one in ***real*** *i*), and its mode is **'integral'**.}

{By a similar mechanism, a **DYADIC-formula** (5.4.2.1.a) may be said to "identify" that **DYADIC-defining-operator** (4.8.1.a) which determines its priority.}

c) The environ E "necessary for" a construct C in an environ E1 is determined as follows:
If E1 is the primal environ {2.2.2.a},
then E is E1;
otherwise, letting E1 be composed of a locale L corresponding to some **'PROPSETY'** and another environ E2,
If C contains any **QUALITY-applied-indicator-with-TAX**

- which does not identify {b} a **defining-indicator** contained in C,
- which is not a **mode-indication** directly descended from a **formal-** or **virtual-declarer**, and
- which is such that the predicate **'where QUALITY TAX resides in PROPSETY'** {7.2.1.b} holds,

then E is E1;

otherwise, {L is not necessary for C and} E is the environ necessary for C in E2.

{The environ necessary for a construct is used in the semantics of **routine-texts** (5.4.1.2) and in "establishing" (3.2.2.b). For example, in

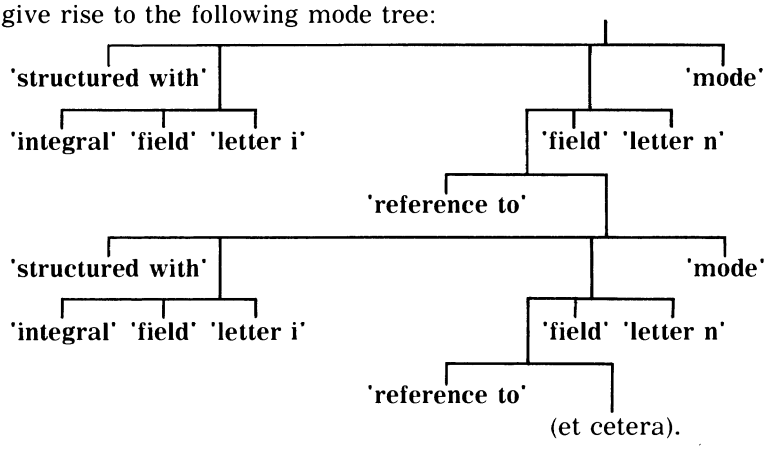
*¢2¢* ***proc void*** *pp;* ***int*** *n; (¢1¢* ***proc*** *p =* ***void*** *: print (n); pp := p)*

if E1 and E2 are the environs established by the elaboration of the **serial-clauses** marked by the **comments** *¢1¢* and *¢2¢*, then E2 is the environ necessary in E1 for the **routine-text** ***void*** *: print (n)*, and so the routine yielded by *p* in E1 is composed of that **routine-text** together with E2 (5.4.1.2). Therefore, the scope of that routine is the scope of E2 (2.1.3.5.c) and hence the assignment (5.2.1.2.b) invoked by *pp := p* is well defined.}

## 7.3. Equivalence of modes

{The equivalence or nonequivalence of '**MOID**'s is determined in this section. For a discussion of equivalent '**MOID**'s see 2.1.1.2.}

{One way of viewing recursive modes is to consider them as infinite trees. Such a "mode tree" is obtained by repeatedly substituting in some spelling, for each '**MU application**', the '**MODE**' of the corresponding '**MU definition of MODE**'. Thus, the spelling '**mui definition of structured with integral field letter i reference to mui application field letter n mode**' would give rise to the following mode tree:




Two spellings are equivalent if and only if they give rise to identical mode trees. The equivalence syntax tests the equivalence of two spellings by, as it were, simultaneously developing the two trees until a difference is found (resulting in a blind alley) or until it becomes apparent that no difference can be found. The growing production tree reflects to some extent the structure of the mode trees.}

### 7.3.1. Syntax

A) **SAFE :: safe ; MU has MODE SAFE ; yin SAFE ; yang SAFE ; remember MOID1 MOID2 SAFE.**

B) **HEAD :: PLAIN ; PREF{71A} ; structured with ;**
**FLEXETY ROWS of ; procedure with ; union of ; void.**

C) **TAILETY :: MOID ; FIELDS mode ; PARAMETERS yielding MOID ;**
**MOODS mode ; EMPTY.**

D) **PARTS :: PART ; PARTS PART.**

E) **PART :: FIELD ; PARAMETER.**

a) **WHETHER MOID1 equivalent MOID2{64b,71m,72c} :**
**WHETHER safe MOID1 equivalent safe MOID2{b}.**

b) **WHETHER SAFE1 MOID1 equivalent SAFE2 MOID2{a,b,e,i,j,n} :**
**where (SAFE1) contains (remember MOID1 MOID2)**
**or (SAFE2) contains (remember MOID2 MOID1),**
**WHETHER true ;**
**unless (SAFE1) contains (remember MOID1 MOID2)**
**or (SAFE2) contains (remember MOID2 MOID1),**
**WHETHER (HEAD3) is (HEAD4)**
**and remember MOID1 MOID2 SAFE3 TAILETY3**
**equivalent SAFE4 TAILETY4{b,d,e,k,q,-},**
**where SAFE3 HEAD3 TAILETY3 develops from**
**SAFE1 MOID1{c}**
**and SAFE4 HEAD4 TAILETY4 develops from**
**SAFE2 MOID2{c}.**

c) **WHETHER SAFE2 HEAD TAILETY develops from**
**SAFE1 MOID{b,c} :**
**where (MOID) is (HEAD TAILETY),**
**WHETHER (HEAD) shields SAFE1 to SAFE2{74a,b,c,d,-} ;**
**where (MOID) is (MU definition of MODE),**
**unless (SAFE1) contains (MU has),**
**WHETHER SAFE2 HEAD TAILETY develops from**
**MU has MODE SAFE1 MODE{c} ;**
**where (MOID) is (MU application)**
**and (SAFE1) is (NOTION MU has MODE SAFE3)**
**and (NOTION) contains (yin) and (NOTION) contains (yang),**
**WHETHER SAFE2 HEAD TAILETY develops from**
**SAFE1 MODE{c}.**

d) **WHETHER SAFE1 FIELDS1 mode**
**equivalent SAFE2 FIELDS2 mode{b} :**
**WHETHER SAFE1 FIELDS1 equivalent SAFE2 FIELDS2{f,g,h,i}.**

e) **WHETHER SAFE1 PARAMETERS1 yielding MOID1**
**equivalent SAFE2 PARAMETERS2 yielding MOID2{b} :**
**WHETHER SAFE1 PARAMETERS1**
**equivalent SAFE2 PARAMETERS2{f,g,h,j}**
**and SAFE1 MOID1 equivalent SAFE2 MOID2{b}.**

f) **WHETHER SAFE1 PARTS1 PART1**
**equivalent SAFE2 PARTS2 PART2{d,e,f} :**
**WHETHER SAFE1 PARTS1 equivalent SAFE2 PARTS2{f,g,h,i,j}**
**and SAFE1 PART1 equivalent SAFE2 PART2{i,j}.**

g) **WHETHER SAFE1 PARTS1 PART1 equivalent**
**SAFE2 PART2{d,e,f} : WHETHER false.**

h) **WHETHER SAFE1 PART1 equivalent**
**SAFE2 PARTS2 PART2{d,e,f} : WHETHER false.**

i) **WHETHER SAFE1 MODE1 field TAG1**
**equivalent SAFE2 MODE2 field TAG2{d,f} :**
**WHETHER (TAG1) is (TAG2)**
**and SAFE1 MODE1 equivalent SAFE2 MODE2{b}.**

j) **WHETHER SAFE1 MODE1 parameter**
**equivalent SAFE2 MODE2 parameter{e,f} :**
**WHETHER SAFE1 MODE1 equivalent SAFE2 MODE2{b}.**

k) **WHETHER SAFE1 MOODS1 mode equivalent**
**SAFE2 MOODS2 mode{b} :**
**WHETHER SAFE1 MOODS1 subset of SAFE2 MOODS2{l,m,n}**
**and SAFE2 MOODS2 subset of SAFE1 MOODS1{l,m,n}**
**and MOODS1 number equals MOODS2 number{o,p}.**

l) **WHETHER SAFE1 MOODS1 MOOD1**
**subset of SAFE2 MOODS2{k,l,46s,64b} :**
**WHETHER SAFE1 MOODS1 subset of SAFE2 MOODS2{l,m,n}**
**and SAFE1 MOOD1 subset of SAFE2 MOODS2{m,n}.**

m) **WHETHER SAFE1 MOOD1**
**subset of SAFE2 MOODS2 MOOD2{k,l,m,46s,64b} :**
**WHETHER SAFE1 MOOD1 subset of SAFE2 MOODS2{m,n}**
**or SAFE1 MOOD1 subset of SAFE2 MOOD2{n}.**

n) **WHETHER SAFE1 MOOD1 subset of SAFE2 MOOD2{k,l,m,64b} :**
**WHETHER SAFE1 MOOD1 equivalent SAFE2 MOOD2{b}.**

o) **WHETHER MOODS1 MOOD1 number equals**
**MOODS2 MOOD2 number{k,o} :**
**WHETHER MOODS1 number equals MOODS2 number{o,p,-}.**

p) **WHETHER MOOD1 number equals MOOD2 number{k,o} :**
**WHETHER true.**

q) **WHETHER SAFE1 EMPTY equivalent SAFE2 EMPTY{b} :**
**WHETHER true.**

{Rule a introduces the **'SAFE'**s which are used as associative memories during the determination of equivalence. There are two of them, one belonging to each mode. Rule b draws an immediate conclusion if the **'MOID'**s under consideration are already remembered (see below) in an appropriate **'SAFE'** in the form **'remember MOID1 MOID2'**. If this is not the case, then the two **'MOID'**s are first remembered in a **'SAFE'** (the one on the left) and then each **'MOID'** is developed (rule c) and split into its **'HEAD'** and its **'TAILETY'**, e.g., **'reference to real'** is split into **'reference to'** and **'real'**.

If the **'HEAD'**s differ, then the matter is settled (rule b); otherwise the **'TAILETY'**s are analyzed according to their structure (which must be the same if the **'HEAD'**s are identical). In each case, except where the

'**HEAD**'s were '**union of**', the equivalence is determined by examining the corresponding components, according to the following scheme:

| rule | '**TAILETY**' | components |
|---|---|---|
| d | '**FIELDS mode**' | '**FIELDS**' |
| e | '**PARAMETERS yielding MOID**' | '**PARAMETERS**' and '**MOID**' |
| f | '**FIELDS FIELD**' | '**FIELDS**' and '**FIELD**' |
| f | '**PARAMETERS PARAMETER**' | '**PARAMETERS**' and '**PARAMETER**' |
| i | '**MODE field TAG**' | '**MODE**' and '**TAG**' |
| j | '**MODE parameter**' | '**MODE**' |

In the case of unions, the '**TAILETY**'s are of the form '**MOODS1 mode**' and '**MOODS2 mode**'. Since '**MOOD**'s within equivalent unions may commute, as in the modes specified by ***union (real, int)*** and ***union (int, real)***, the equivalence is determined by checking that '**MOODS1**' is a subset of '**MOODS2**' and that '**MOODS2**' is a subset of '**MOODS1**', where the subset test, of course, invokes the equivalence test recursively (rules k,l,m,n,o,p).

A '**MOID**' is developed (rule c) into the form '**HEAD TAILETY**' by determining that
(i) it is already of that form: in which case markers ('**yin**' and '**yang**') may be placed in its '**SAFE**' for the later determination of well-formedness (see 7.4);
(ii) it is some '**MU definition of MODE**': in which case '**MU has MODE**' is stored in its '**SAFE**' (provided that this particular '**MU**' is not there already) and the '**MODE**' is developed;
(iii) it is some '**MU application**': in which case there must be some '**MU has MODE**' in its '**SAFE**' already. That '**MODE**' is then developed after a well-formedness check (see 7.4) consisting of the determination that there is at least one '**yin**' and at least one '**yang**' in the '**SAFE**' which is more recent than the '**MU has MODE**'.}

{Before a pair of '**TAILETY**'s is tested for equivalence, it is remembered in the '**SAFE**' that the original pair of '**MOID**'s is being tested. This is used to force a shortcut to '**WHETHER true**' if these '**MOID**'s should ever be tested again for equivalence lower down the production tree. Since the number of pairs of component '**MOID**'s that can be derived from any two given '**MOID**'s is finite, it follows that the testing process terminates.

It remains to be shown that the process is correct. Consider the unrestricted (possibly infinite) production tree that would be obtained if there were no shortcut in the syntax (by omitting the first alternative together with the first member of the other alternative of rule b). If two '**MOID**'s are not equivalent, then there exists in their mode trees a shortest path from the top node to some node exhibiting a difference. Obviously, the reflection of this shortest path in the unrestricted production tree cannot contain a repeated test for the equivalence of any pair of '**MOID**'s, and therefore none of the shortcuts to '**WHETHER true**' in

the restricted production tree can occur on this shortest path. Consequently, the path to the difference must be present also in the (restricted) production tree produced by the syntax. If the testing process does not exhibit a difference in the restricted tree, then no difference can be found in any number of steps; i.e., the '**MOID**'s are equivalent.}

7.4. Well-formedness

{A mode is well formed if

(i) the elaboration of an **actual-declarer** specifying that mode is a finite action (i.e., any value of that mode can be stored in a finite memory) and

(ii) it is not strongly coercible from itself (since this would lead to ambiguities in coercion).}

7.4.1. Syntax

a) **WHETHER (NOTION) shields SAFE to SAFE{73c} :**
**where (NOTION) is (PLAIN)**
**or (NOTION) is (FLEXETY ROWS of)**
**or (NOTION) is (union of) or (NOTION) is (void),**
**WHETHER true.**

b) **WHETHER (PREF) shields SAFE to yin SAFE{73c} : WHETHER true.**

c) **WHETHER (structured with) shields SAFE to yang SAFE{73c} :**
**WHETHER true.**

d) **WHETHER (procedure with) shields SAFE to yin yang SAFE{73c} :**
**WHETHER true.**

{As a by-product of mode equivalencing, modes are tested for well-formedness (7.3.1.c). All nonrecursive modes are well formed. For recursive modes, it is necessary that each cycle in each spelling of that mode (from '**MU definition of MODE**' to '**MU application**') passes through at least one '**HEAD**' which is yin, ensuring condition (i) and one (possibly the same) '**HEAD**' which is yang, ensuring condition (ii). Yin '**HEAD**'s are '**PREF**' and '**procedure with**'. Yang '**HEAD**'s are '**structured with**' and '**procedure with**'. The other '**HEAD**'s, including '**FLEXETY ROWS of**' and '**union of**', are neither yin nor yang. This means that the modes specified by ***a***, ***b*** and ***c*** in

***mode a*** = ***struct (int*** *n*, ***ref a*** *next*), ***b*** = ***struct (proc b*** *next*), ***c*** = ***proc (c) c***

are all well formed. However, ***mode d*** = [*1* : *10*] ***d***, ***e*** = ***union (int, e)*** is not a **mode-declaration**.}

{Tao produced the one.
The one produced the two.
The two produced the three.
And the three produced the ten thousand things.
The ten thousand things carry the yin and embrace the yang, and through the blending of the material force they achieve harmony.
Tao-te Ching, 42, Lao Tzu.}

# PART IV

## Elaboration-independent constructions

### 8. Denotations

{**Denotations**, e.g., *3.14* or *"abc"*, are constructs whose yields are independent of any action. In other languages, they are sometimes termed "literals" or "constants".}

8.0.1. Syntax

a) **MOID NEST denoter**{5D,A341i} : **pragment**{92a} **sequence option,**
**MOID denotation**{810a,811a,812a,813a,814a,815a,82a,b,c,83a,-}.

{The meaning of a **denotation** is independent of any nest.}

8.1. Plain denotations

{**Plain-denotations** are those of arithmetic values, truth values, characters and the void value, e.g., *1*, *3.14*, ***true***, *"a"* and ***empty***.}

8.1.0.1. Syntax

A) **SIZE :: long ; short.**
B) ***NUMERAL :: fixed point numeral ; variable point numeral ;**
**floating point numeral.**

a) **SIZE INTREAL denotation**{a,80a} :
**SIZE symbol**{94d}, **INTREAL denotation**{a,811a,812a}.

b) ***plain denotation : PLAIN denotation**{a,811a,812a,813a,814a} ;
**void denotation**{815a}.

{Example:

a) ***long 0*** }

8.1.0.2. Semantics

The yield W of an **INTREAL-denotation** is the "intrinsic value" {8.1.1.2, 8.1.2.2.a,b} of its constituent **NUMERAL**;
• it is required that W be not greater than the largest value of mode **'INTREAL'** that can be distinguished {2.1.3.1.d}.

{An **INTREAL-denotation** yields an arithmetic value {2.1.3.1.a}, but arithmetic values yielded by different **INTREAL-denotations** are not necessarily different (e.g., *123.4* and $1.234_{10}+2$). }

8.1.1. Integral denotations

8.1.1.1. Syntax

a) **integral denotation**{80a,810a} : **fixed point numeral**{b}.
b) **fixed point numeral**{a,812c,d,f,i,A341h} : **digit cypher**{c} **sequence.**
c) **digit cypher**{b} : **DIGIT symbol**{94b}.

{Examples:

a) *4096* b) *4096*
c) *4* }

8.1.1.2. Semantics

The intrinsic value of a **fixed-point-numeral** N is the integer of which the reference-language form of N {9.3.b} is a decimal representation.

8.1.2. Real denotations

8.1.2.1. Syntax

a) **real denotation**{80a,810a} :
**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} : **fixed point numeral**{811b}.
d) **fractional part**{b} : **point symbol**{94b}, **fixed point numeral**{811b}.
e) **floating point numeral**{a} : **stagnant part**{f}, **exponent part**{g}.
f) **stagnant part**{e} :
**fixed point numeral**{811b} ; **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**{94b} ; **letter e symbol**{94a}.
i) **power of ten**{g} : **plusminus**{j} **option**, **fixed point numeral**{811b}.
j) **plusminus**{i} : **plus symbol**{94c} ; **minus symbol**{94c}.

{Examples:

a) *0.00123* • *1.23e-3* b) *0.00123*
c) *0* d) *.00123*
e) *1.23e-3* f) *123* • *1.23*
g) *e-3* h) $_{10}$ • *e*
i) *-3* j) + • - }

8.1.2.2. Semantics

a) The intrinsic value V of a **variable-point-numeral** N is determined as follows:
• let I be the intrinsic value of the **fixed-point-numeral** of its constituent **integral-part**, if any, and be 0 otherwise;
• let F be the intrinsic value of the **fixed-point-numeral** of its **fractional-part** P divided by 10 as many times as there are **digit-cyphers** contained in P;
• V is the sum in the sense of numerical analysis of I and F.

b) The intrinsic value V of a **floating-point-numeral** N is determined as follows:

• let S be the intrinsic value of the **NUMERAL** of its **stagnant-part**;
• let E be the intrinsic value of the constituent **fixed-point-numeral** of its **exponent-part**;
Case A: The constituent **plusminus-option** of N contains a **minus-symbol**:
• V is the product in the sense of numerical analysis of S and 1/10 raised to the power E;
Case B: The direct descendent of that **plusminus-option** contains a **plus-symbol** or is empty:
• V is the product in the sense of numerical analysis of S and 10 raised to the power E.

8.1.3. Boolean denotations

8.1.3.1. Syntax

a) **boolean denotation**{80a} : **true**{94b} **symbol** ; **false**{94b} **symbol.**

{Examples:

a) ***true*** • ***false*** }

8.1.3.2. Semantics

The yield of a **boolean-denotation** is true (false) if its direct descendent is a **true-symbol** (**false-symbol**).

8.1.4. Character denotations

{**Character-denotations** consist of a **string-item** between two **quote-symbols**, e.g., *"a"*. To indicate a quote, a **quote-image-symbol** (represented by *""*) is used, e.g., *""""*. Since the syntax nowhere allows **character-** or **string-denotations** to follow one another, this causes no ambiguity.}

8.1.4.1. Syntax

a) **character denotation**{80a} :
**quote**{94b} **symbol, string item**{b}, **quote symbol**{94b}.
b) **string item**{a,83b} : **character glyph**{c} ; **quote image symbol**{94b} ;
**other string item**{d}.
c) **character glyph**{b,92c} : **LETTER symbol**{94a} ;
**DIGIT symbol**{94b} ; **point symbol**{94b} ; **open symbol**{94f} ;
**close symbol**{94f} ; **comma symbol**{94b} ; **space symbol**{94b} ;
**plus symbol**{94c} ; **minus symbol**{94c}.
d) A production rule may be added for the notion **'other string item'** {b, for which no hyper-rule is given in this Report} each of whose alternatives is a symbol {1.1.3.1.f} which is different from any terminal production of **'character glyph'** {c} and which is not **'quote symbol'**.

{Examples:

a) *"a"* b) *a* • *""* • ?
c) *a* • *1* • . • *(* • *)* • , • *+* • *-* }

8.1.4.2. Semantics

a) The yield of a **character-denotation** is the intrinsic value of the **symbol** descended from its **string-item**.

b) The intrinsic value of each distinct **symbol** descended from a **string-item** is a unique character. {Characters have no inherent meaning, except insofar as some of them are interpreted in particular ways by the transput declarations (10.3). The **character-glyphs**, which include all the characters needed for transput, form a minimum set which all implementations (2.2.2.c) are expected to provide.}

8.1.5. Void denotation

{A **void-denotation** may be used to assign a void value to a **UNITED-variable**, e.g., ***union*** *(*[ ]***real, void***) u* := ***empty***.}

8.1.5.1. Syntax

a) **void denotation**{80a} : **empty**{94b} **symbol.**

{Example:

a) ***empty*** }

8.1.5.2. Semantics

The yield of a **void-denotation** is empty.

8.2. Bits denotations

8.2.1. Syntax

A) **RADIX :: radix two ; radix four ; radix eight ; radix sixteen.**

a) **structured with row of boolean field**
**LENGTH LENGTHETY letter aleph mode denotation**{a,80a} :
**long**{94d} **symbol, structured with row of boolean field**
**LENGTHETY letter aleph mode denotation** {a,c}.

b) **structured with row of boolean field**
**SHORTH SHORTHETY letter aleph mode denotation**{b,80a} :
**short**{94d} **symbol, structured with row of boolean field**
**SHORTHETY letter aleph mode denotation**{b,c}.

c) **structured with row of boolean field**
**letter aleph mode denotation**{a,b,80a} :
**RADIX**{d,e,f,g}, **letter r symbol**{94a}, **RADIX digit**{h,i,j,k} **sequence.**

d) **radix two**{c,A347b} : **digit two**{94b} **symbol.**

e) **radix four**{c,A347b} : **digit four**{94b} **symbol.**

f) **radix eight**{c,A347b} : **digit eight**{94b} **symbol.**

g) **radix sixteen**{c,A347b} : **digit one symbol**{94b}, **digit six symbol**{94b}.

h) **radix two digit**{c,i} : **digit zero symbol**{94b} ; **digit one symbol**{94b}.

i) **radix four digit**{c,j} : **radix two digit**{h} ; **digit two symbol**{94b} ;
**digit three symbol**{94b}.

j) **radix eight digit**{c,k} : **radix four digit**{i} ; **digit four symbol**{94b} ;
**digit five symbol**{94b} ; **digit six symbol**{94b} ;
**digit seven symbol**{94b}.

k) **radix sixteen digit**{c} : **radix eight digit**{j} ; **digit eight symbol**{94b} ;
**digit nine symbol**{94b} ; **letter a symbol**{94a} ;
**letter b symbol**{94a} ; **letter c symbol**{94a} ; **letter d symbol**{94a} ;
**letter e symbol**{94a} ; **letter f symbol**{94a}.

l) ***bits denotation : BITS denotation**{a,b,c}.
{**BITS :: structured with**
**row of boolean field SITHETY letter aleph mode.**}

m) ***radix digit : RADIX digit**{h,i,j,k}.

{Examples:

a) ***long*** *2r101* b) ***short*** *16rffff*
c) *8r231* }

8.2.2. Semantics

a) The yield V of a **bits-denotation** D is determined as follows:
• let W be the intrinsic boolean value {b} of its constituent **RADIX-digit-sequence**;
• let m be the length of W;
• let n be the value of *L bits width* {10.2.1.j}, where *L* stands for as many times *long* (*short*) as there are **long-symbols** (**short-symbols**) contained in D;
• it is required that m be not greater than n;
• V is a structured value {whose mode is some '**BITS**'} whose only field is a multiple value having
(i) a descriptor ((1, n)) and
(ii) n elements, that selected by (i) being false if $1 \leq i \leq n - m$, and being the (i + m − n)-th truth value of {the sequence} W otherwise.

b) The intrinsic boolean value of a **RADIX-digit-sequence** S is the shortest sequence of truth values which, regarded as a binary number (true corresponding to 1 and false to 0), is the same as the intrinsic integral value {c} of S.

c) The intrinsic integral value of a **radix-two-** (**radix-four-**, **radix-eight-**, **radix-sixteen-**) **-digit-sequence** S is the integer of which the reference-language form of S {9.3.b} is a binary, (quaternary, octal, hexadecimal) representation, where the representations *a*, *b*, *c*, *d*, *e* and *f*, considered as digits, have values 10, 11, 12, 13, 14 and 15 respectively.

8.3. String denotations

{**String-denotations** are a convenient way of specifying "strings", i.e., multiple values of mode '**row of character**'.
Example:
***string*** *message* := *"all is well"* }

8.3.1. Syntax

a) **row of character denotation**{80a} :
**quote**{94b} **symbol, string**{b} **option, quote symbol**{94b}.

b) **string**{a} : **string item**{814b}, **string item**{814b} **sequence.**

c) ***string denotation : row of character denotation**{a}.

{Examples:

a) *"abc"* b) *abc* }

8.3.2. Semantics

The yield of a **string-denotation** D is a multiple value V determined as follows:

- let n be the number of **string-items** contained in D;
- the descriptor of V is ((1, n));
- for i = 1, ... , n, the element of V with index (i) is the intrinsic value {8.1.4.2.b} of the i-th constituent **symbol** of the **string** of D.

{*"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 (6.6). Elsewhere, where a multiple value is required, a **cast** (5.5.1.1.a) may be used, e.g., ***union (char, string)*** *cs* := ***string*** *("a").*}

## 9. Tokens and symbols

9.1. Tokens

{**Tokens** (9.1.1.f) are **symbols** (9.1.1.h) possibly preceded by **pragments** (9.2.1.a). Therefore, **pragments** may appear between **symbols** wherever the syntax produces a succession of **tokens**. However, in a few places, the syntax specifically produces **symbols** rather than **tokens**, notably within **denotations** (8), **format-texts** (10.3.4.1.1.a) and, of course, within **pragments**. Therefore, **pragments** may not occur in these places.}

9.1.1. Syntax

a) **CHOICE STYLE start**{34a} :
**where (CHOICE) is (choice using boolean),**
**STYLE if**{94f,-} **token ;**
**where (CHOICE) is (CASE), STYLE case**{94f,-} **token.**

b) **CHOICE STYLE in**{34e} :
**where (CHOICE) is (choice using boolean),**
**STYLE then**{94f,-} **token ;**
**where (CHOICE) is (CASE), STYLE in**{94f,-} **token.**

c) **CHOICE STYLE again**{34l} :
**where (CHOICE) is (choice using boolean),**
**STYLE else if**{94f,-} **token ;**
**where (CHOICE) is (CASE), STYLE ouse**{94f,-} **token.**

d) **CHOICE STYLE out**{34l} :
**where (CHOICE) is (choice using boolean),**
**STYLE else**{94f,-} **token ;**
**where (CHOICE) is (CASE), STYLE out**{94f,-} **token.**

e) **CHOICE STYLE finish**{34a} :
**where (CHOICE) is (choice using boolean),**
**STYLE fi**{94f,-} **token ;**
**where (CHOICE) is (CASE), STYLE esac**{94f,-} **token.**

f) **NOTION token : pragment**{92a} **sequence option,**
**NOTION symbol**{94a,b,c,d,e,f,g,h}.

g) ***token : NOTION token**{f}.

h) ***symbol : NOTION symbol**{94a,b,c,d,e,f,g,h}.

9.2. Comments and pragmats

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

{A **pragment** is a **comment** or a **pragmat**. No semantics of **pragments** is given and therefore the meaning (2.1.4.1.a) of any **program** is quite unaffected by their presence. It is indeed the intention that **comments** should be entirely ignored by the implementation, their sole purpose being the enlightenment of the human interpreter of the **program**.

**Pragmats** may, on the other hand, convey to the implementation some piece of information affecting some aspect of the meaning of the **program** which is not defined by this Report, for example:

- the action to be taken upon overflow (2.1.4.3.h) or if the scope rule is violated (as in 5.2.1.2.b), e.g., ***pr** overflow check on **pr**, **pr** overflow check off **pr**, **pr** scope check on **pr*** or ***pr** scope check off **pr***;
- the action to be taken upon completion of the compilation process, e.g., ***pr** compile only **pr**, **pr** dump **pr*** or ***pr** run **pr***;
- that the language to be implemented is some sublanguage or superlanguage of ALGOL 68, e.g., ***pr** nonrec **pr*** (for a **routine-text** which may be presumed to be non-recursive);
- that the compilation may check for the truth, or attempt to prove the correctness, of some assertion, e.g.:

***int** a, b; read((a, b)) **pr assert** $a \geq 0 \wedge b > 0$ **pr**;*
***int** q := 0, r := a;*
***while** $r \geq b$ **pr assert** $a = b \times q + r \wedge 0 \leq r$ **pr***
***do** (q +:= 1, r −:= b) **od***
***pr assert** $a = b \times q + r \wedge 0 \leq r \wedge r < b$ **pr** .*

They may also be used to convey to the implementation that the source text is to be augmented with some other text, or edited in some way, for example:

- some previously compiled portion of the **particular-program** is to be invoked, e.g., ***pr with*** *segment* ***from*** *album* ***pr***;
- the source text is continued on some other document, e.g., ***pr read from*** *another file* ***pr***;
- the end of the source text has been reached, e.g., ***pr finish pr***.

The interpretation of **pragmats** is not defined in this Report, but is left to the discretion of the implementer, who ought, at least, to provide some means whereby all further **pragmats** may be ignored, for example:

***pr*** *pragmats off* ***pr***.}

```
{pr algol 68 pr
begin
 proc pr nonrec pr pr = void: pr;
 pr
end
pr run pr pr ? pr
Revised Report on the Algorithmic
                        Language ALGOL 68.}
```

9.2.1. Syntax

A) **PRAGMENT :: pragmat ; comment.**

a) **pragment**{80a,91f,A341b,h,A348a,b,c,A349a,A34Ab} **: PRAGMENT**{b}.

b) **PRAGMENT**{a} **: STYLE PRAGMENT symbol**{94h,-},
**STYLE PRAGMENT item**{c} **sequence option**,
**STYLE PRAGMENT symbol**{94h,-}.
{**STYLE :: brief ; bold ; style TALLY.**}

c) **STYLE PRAGMENT item**{b} **: character glyph**{814c} ;
**STYLE other PRAGMENT item**{d}.

d) A production rule may be added for each notion designated by '**STYLE other PRAGMENT item**' {c, for which no hyper-rule is given in this Report} each of whose alternatives is a symbol {1.1.3.1.f}. different from any terminal production of '**character glyph**' {8.1.4.1.c}, and such that no terminal production of any '**STYLE other PRAGMENT item**' is the corresponding '**STYLE PRAGMENT symbol**'. {Thus ***comment*** ¢ ***comment*** might be a **comment**, but ¢ ¢ ¢ could not.}

{Examples:

a) ***pr*** *list* ***pr*** • ¢ *source program to be listed* ¢
c) *l* • ? }

9.3. Representations

a) A construct in the strict language must be represented in some "representation language" such as the "reference language", which is used in this Report. Other representation languages specially suited to the

supposed preference of some human or mechanical interpreter **of** the language may be termed "publication" or "hardware" languages. {The reference language is intended to be used for the representation of **particular-programs** and of their descendents. It is, however, also used in Chapter 10 for the definition of the standard environment.}

b) A "construct in a representation language" is obtained from the terminal production T {1.1.3.2.f} of the corresponding construct in the strict language {1.1.3.2.e} by replacing the **symbols** in T by their representations, as specified in 9.4 below in the case of the reference language.

{Thus, the strict-language **particular-program** whose terminal production is

**'bold begin symbol' 'skip symbol' 'bold end symbol'**

gives rise to the reference language **particular-program**

***begin skip end*** .}

c) An implementation {2.2.2.c} of ALGOL 68 which uses representations which are sufficiently close to those of the reference language to be recognized without further elucidation, and which does not augment or restrict the available representations other than as provided for below {9.4.a,b,c}, is an "implementation of the reference language".

{E.g., ***begin***, *begin*, *BEGIN*, *'begin* and *'begin'* could all be representations of the **bold-begin-symbol** in an implementation of the reference language; some combination of holes in a punched card might be a representation of it in some hardware language.}

## 9.4. The reference language

a) The reference language provides representations for various **symbols**, including an arbitrarily large number of **TAX-symbols** {where **TAX :: TAG ; TAB ; TAD ; TAM.**}. The representations of some of them are specified below {9.4.1}, and to these may be added suitable representations for **style-TALLY-letter-ABC-symbols** and **style-TALLY-monad-symbols** and any terminal productions of **'STYLE other PRAGMENT item'** {9.2.1.d} and of **'other string item'** {8.1.4.1.d}. Representations are not provided for any of these {but they enable individual implementations to make available their full character sets for use as characters, to provide additional or extended alphabets for the construction of **TAG-** and **TAB-symbols**, and to provide additional **symbols** for use as **operators**}. There is not, however, {and there must not be,} except in representations of the **standard-**, and other, **preludes** {10.1.3.Step 6}, any representation of the **letter-aleph-symbol** or the **primal-symbol**. {For the remaining **TAX-symbols**, see 9.4.2. There are, however, some **symbols** produced by the syntax, e.g., the **brief-pragmat-symbol**, for which no representation is provided at all. This does not preclude the representation of such **symbols** in other representation languages.}

b) Where more than one representation of a **symbol** is given, any of them may be chosen. Moreover, it is sufficient for an implementation of the reference language to provide only one. Also, it is not necessary to provide a representation of any particular **MONAD-symbol** or **NOMAD-symbol** so long as those that are provided are sufficient to represent at least one version {10.1.3.Step 3} of each **operator** declared in the **standard-prelude**.

{For certain different **symbols**, one same or nearly the same representation is given; e.g., the representation ":" is given for the **routine-symbol**, the **colon-symbol** and the **up-to-symbol** and ":" for the **label-symbol**. It follows uniquely from the syntax which of these four **symbols** is represented by an occurrence, outside **comments**, **pragmats** and **string-denotations**, of any mark similar to either of those representations. It is also the case that ".." could be used, without ambiguity, for any of them, and such might indeed be necessary in implementations with limited character sets. It may be noted that, for such implementations, no ambiguity would be introduced were "(/" and "/)" to be used as representations of the **style-ii-sub-symbol** and the **style-ii-bus-symbol**, respectively.

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 **routine-symbol**, etc., and "=", the representation of the **equals-symbol** and of the **is-defined-as-symbol**. It follows from the syntax that ":=" can occur, outside **comments**, **pragmats** and **string-denotations**, as a representation of the **becomes-symbol** only (since "=" cannot occur as the representation of a **monadic-operator**). Similarly, the other given composite representations do not cause ambiguity.}

c) The fact that the representations of the **letter-ABC-symbols** given {9.4.1.a} are usually spoken of as small letters is not meant to imply that the corresponding capital letters could not serve equally well. {On the other hand, if both a small letter and the corresponding capital letter occur, then one of them is presumably the representation of some **style-TALLY-letter-ABC-symbol** or of a **bold-letter-ABC-symbol**. See also 1.1.5.b for the possibility of additional '**ABC**'s in a variant of the language.}

d) A "typographical display feature" is a blank, or a change to a new line or a new page. Such features, when they appear between the **symbols** of a construct in the reference language, are of no significance and do not affect the meaning of that construct. However, a blank contained within a **string-** or **character-denotation** is one of the representations of the **space-symbol** {9.4.1.b} rather than a typographical display feature. Where the representation of a **symbol** in the reference language is composed of several marks (e.g., ***to***, :=), those marks form one {indivisible} **symbol** and, unless the contrary is explicitly stated {9.4.2.2.a,c}, typographical display features may not separate them.

9.4.1. Representations of symbols

a) Letter symbols

| symbol | representation |
|---|---|
| **letter a symbol**{814c,82k,942B,A346b} | *a* |
| **letter b symbol**{814c,82k,942B,A344b} | *b* |
| **letter c symbol**{814c,82k,942B,A348a} | *c* |
| **letter d symbol**{814c,82k,942B,A342f} | *d* |
| **letter e symbol**{812h,814c,82k,942B,A343e} | *e* |
| **letter f symbol**{814c,82k,942B,A349a} | *f* |
| **letter g symbol**{814c,942B,A34Aa} | *g* |
| **letter h symbol**{814c,942B} | *h* |
| **letter i symbol**{814c,942B,A345b} | *i* |
| **letter j symbol**{814c,942B} | *j* |
| **letter k symbol**{814c,942B,A341f} | *k* |
| **letter l symbol**{814c,942B,A341f} | *l* |
| **letter m symbol**{814c,942B} | *m* |
| **letter n symbol**{814c,942B,A341h} | *n* |
| **letter o symbol**{814c,942B} | *o* |
| **letter p symbol**{814c,942B,A341f} | *p* |
| **letter q symbol**{814c,942B,A341f} | *q* |
| **letter r symbol**{814c,82c,942B,A347c} | *r* |
| **letter s symbol**{814c,942B,A341l} | *s* |
| **letter t symbol**{814c,942B} | *t* |
| **letter u symbol**{814c,942B} | *u* |
| **letter v symbol**{814c,942B} | *v* |
| **letter w symbol**{814c,942B} | *w* |
| **letter x symbol**{814c,942B,A341f} | *x* |
| **letter y symbol**{814c,942B,A341f} | *y* |
| **letter z symbol**{814c,942B,A342d} | *z* |

b) Denotation symbols

| symbol | representation |
|---|---|
| **digit zero symbol**{811c,814c,82h,942C} | *0* |
| **digit one symbol**{43b,811c,814c,82g,h,942C} | *1* |
| **digit two symbol**{43b,811c,814c,82d,i,942C} | *2* |
| **digit three symbol**{43b,811c,814c,82i,942C} | *3* |
| **digit four symbol**{43b,811c,814c,82e,j,942C} | *4* |
| **digit five symbol**{43b,811c,814c,82j,942C} | *5* |
| **digit six symbol**{43b,811c,814c,82g,j,942C} | *6* |
| **digit seven symbol**{43b,811c,814c,82j,942C} | *7* |
| **digit eight symbol**{43b,811c,814c,82f,k,942C} | *8* |
| **digit nine symbol**{43b,811c,814c,82k,942C} | *9* |
| **point symbol**{812d,814c,A343d} | . |
| **times ten to the power symbol**{812h} | *10* \ |

| | |
|---|---|
| **true symbol**{813a} | ***true*** |
| **false symbol**{813a} | ***false*** |
| **quote symbol**{814a,83a} | " |
| **quote image symbol**{814b} | "" |
| **space symbol**{814c} | |
| **comma symbol**{814c} | , |
| **empty symbol**{815a} | ***empty*** |

c) Operator symbols

| symbol | representation |
|---|---|
| **or symbol**{942H} | ∨ |
| **and symbol**{942H} | ∧ |
| **ampersand symbol**{942H} | & |
| **differs from symbol**{942H} | ≠ |
| **is less than symbol**{942I} | < |
| **is at most symbol**{942H} | ≤ |
| **is at least symbol**{942H} | ≥ |
| **is greater than symbol**{942I} | > |
| **divided by symbol**{942I} | / |
| **over symbol**{942H} | ÷ |
| **percent symbol**{942H} | **%** |
| **window symbol**{942H} | □ |
| **floor symbol**{942H} | ⌊ |
| **ceiling symbol**{942H} | ⌈ |
| **plus i times symbol**{942H} | ⊥ |
| **not symbol**{942H} | ¬ |
| **tilde symbol**{942H} | ~ |
| **down symbol**{942H} | ↓ |
| **up symbol**{942H} | ↑ |
| **plus symbol**{812j,814c,942H,A342e} | + |
| **minus symbol**{812j,814c,942H,A342e} | − |
| **equals symbol**{942I} | = |
| **times symbol**{942I} | × |
| **asterisk symbol**{942I} | * |
| **assigns to symbol**{942J} | =: |
| **becomes symbol**{44f,521a,942J} | := |

d) Declaration symbols

| symbol | representation |
|---|---|
| **is defined as symbol**{42b,43b,44c,45c} | = |
| **long symbol**{810a,82a} | ***long*** |
| **short symbol**{810a,82b} | ***short*** |
| **reference to symbol**{46c} | ***ref*** |
| **local symbol**{523a,b} | ***loc*** |
| **heap symbol**{523a,b} | ***heap*** |

| | |
|---|---|
| **structure symbol**{46d} | ***struct*** |
| **flexible symbol**{46g} | ***flex*** |
| **procedure symbol**{44b,46o} | ***proc*** |
| **union of symbol**{46s} | ***union*** |
| **operator symbol**{45a} | ***op*** |
| **priority symbol**{43a} | ***prio*** |
| **mode symbol**{42a} | ***mode*** |

e) Mode standards

| symbol | representation |
|---|---|
| **integral symbol**{942E} | ***int*** |
| **real symbol**{942E} | ***real*** |
| **boolean symbol**{942E} | ***bool*** |
| **character symbol**{942E} | ***char*** |
| **format symbol**{942E} | ***format*** |
| **void symbol**{942E} | ***void*** |
| **complex symbol**{942E} | ***compl*** |
| **bits symbol**{942E} | ***bits*** |
| **bytes symbol**{942E} | ***bytes*** |
| **string symbol**{942E} | ***string*** |
| **sema symbol**{942E} | ***sema*** |
| **file symbol**{942E} | ***file*** |
| **channel symbol**{942E} | ***channel*** |

f) Syntactic symbols

| symbol | representation |
|---|---|
| **bold begin symbol**{133d} | ***begin*** |
| **bold end symbol**{133d} | ***end*** |
| **brief begin symbol**{133d,A348b,A34Ab} | *(* |
| **brief end symbol**{133d,A348b,A34Ab} | *)* |
| **and also symbol**{133c,33b,f,34h,41a,b,46e,i, q,t,532b,541e,543b,A348b,A34Ac,d} | , |
| **go on symbol**{32b} | ; |
| **completion symbol**{32b} | ***exit*** |
| **label symbol**{32c} | *:* |
| **parallel symbol**{33c} | ***par*** |
| **open symbol**{814c} | *(* |
| **close symbol**{814c} | *)* |
| **bold if symbol**{91a} | ***if*** |
| **bold then symbol**{91b} | ***then*** |
| **bold else if symbol**{91c} | ***elif*** |
| **bold else symbol**{91d} | ***else*** |
| **bold fi symbol**{91e} | ***fi*** |
| **bold case symbol**{91a} | ***case*** |
| **bold in symbol**{91b} | ***in*** |

| | | | |
|---|---|---|---|
| **bold ouse symbol**{91c} | ***ouse*** | | |
| **bold out symbol**{91d} | ***out*** | | |
| **bold esac symbol**{91e} | ***esac*** | | |
| **brief if symbol**{91a} | ( | | |
| **brief then symbol**{91b} | \| | | |
| **brief else if symbol**{91c} | \|: | | |
| **brief else symbol**{91d} | \| | | |
| **brief fi symbol**{91e} | ) | | |
| **brief case symbol**{91a} | ( | | |
| **brief in symbol**{91b} | \| | | |
| **brief ouse symbol**{91c} | \|: | | |
| **brief out symbol**{91d} | \| | | |
| **brief esac symbol**{91e} | ) | | |
| **colon symbol**{34j,k} | : | | |
| **brief sub symbol**{133e} | [ | | |
| **brief bus symbol**{133e} | ] | | |
| **style i sub symbol**{133e} | ( | | |
| **style i bus symbol**{133e} | ) | | |
| **up to symbol**{46j,k,l,532f} | : | | |
| **at symbol**{532g} | @ | ***at*** | |
| **is symbol**{522b} | :=: | ***is*** | |
| **is not symbol**{522b} | :≠: | :/=: | ***isnt*** |
| **nil symbol**{524a} | ∘ | ***nil*** | |
| **of symbol**{531a} | ***of*** | | |
| **routine symbol**{541a,b} | : | | |
| **bold go to symbol**{544b} | ***goto*** | | |
| **bold go symbol**{544b} | ***go*** | | |
| **skip symbol**{552a} | ~ | ***skip*** | |
| **formatter symbol**{A341a} | $ | | |

g) Loop symbols

| symbol | representation |
|---|---|
| **bold for symbol**{35b} | ***for*** |
| **bold from symbol**{35d} | ***from*** |
| **bold by symbol**{35d} | ***by*** |
| **bold to symbol**{35d,544b} | ***to*** |
| **bold while symbol**{35g} | ***while*** |
| **bold do symbol**{35h} | ***do*** |
| **bold od symbol**{35h} | ***od*** |

h) Pragment symbols

| symbol | representation |
|---|---|
| **brief comment symbol**{92b} | ¢ |
| **bold comment symbol**{92b} | ***comment*** |
| **style i comment symbol**{92b} | ***co*** |

| | |
|---|---|
| style ii comment symbol{92b} | # |
| bold pragmat symbol{92b} | ***pragmat*** |
| style i pragmat symbol{92b} | ***pr*** |

9.4.2. Other TAX symbols

9.4.2.1. Metasyntax

A) **TAG**{D,F,K,48a,b,c,d} ::
**LETTER**{B} ; **TAG LETTER**{B} ; **TAG DIGIT**{C}.

B) **LETTER**{A} ::
**letter ABC**{94a} ; **letter aleph**{-} ; **style TALLY letter ABC**{-}.

C) **DIGIT**{A} :: **digit zero**{94b} ; **digit one**{94b} ; **digit two**{94b} ;
**digit three**{94b} ; **digit four**{94b} ; **digit five**{94b} ; **digit six**{94b} ;
**digit seven**{94b} ; **digit eight**{94b} ; **digit nine**{94b}.

D) **TAB**{48a,b} :: **bold TAG**{A,-} ; **SIZETY STANDARD**{E}.

E) **STANDARD**{D} :: **integral**{94e} ; **real**{94e} ; **boolean**{94e} ;
**character**{94e} ; **format**{94e} ; **void**{94e} ; **complex**{94e} ; **bits**{94e} ;
**bytes**{94e} ; **string**{94e} ; **sema**{94e} ; **file**{94e} ; **channel**{94e}.

F) **TAD**{48a,b} :: **bold TAG**{A,-} ; **DYAD**{G} **BECOMESETY**{J} ;
**DYAD**{G} **cum NOMAD**{I} **BECOMESETY**{J}.

G) **DYAD**{F} :: **MONAD**{H} ; **NOMAD**{I}.

H) **MONAD**{G,K} :: **or**{94c} ; **and**{94c} ; **ampersand**{94c} ;
**differs from**{94c} ; **is at most**{94c} ; **is at least**{94c} ; **over**{94c} ;
**percent**{94c} ; **window**{94c} ; **floor**{94c} ; **ceiling**{94c} ;
**plus i times**{94c} ; **not**{94c} ; **tilde**{94c} ; **down**{94c} ; **up**{94c} ;
**plus**{94c} ; **minus**{94c} ; **style TALLY monad**{-}.

I) **NOMAD**{F,G,K} :: **is less than**{94c} ; **is greater than**{94c} ;
**divided by**{94c} ; **equals**{94c} ; **times**{94c} ; **asterisk**{94c}.

J) **BECOMESETY**{F,K} :: **cum becomes**{94c} ; **cum assigns to**{94c} ;
**EMPTY**.

K) **TAM**{48a,b} :: **bold TAG**{A,-} ; **MONAD**{H} **BECOMESETY**{J} ;
**MONAD**{H} **cum**{9422e} **NOMAD**{I} **BECOMESETY**{J}.

L) **ABC**{B} :: **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**.

M) ***DOP** :: **DYAD**{G} ; **DYAD**{G} **cum NOMAD**{I}.

{The metanotion "**ABC**" is provided, in addition to the metanotion "**ALPHA**", in order to facilitate the definition of variants of ALGOL 68 (1.1.5.b).}

9.4.2.2. Representation

a) The representation of each **TAG-symbol** not given above {9.4.1} is composed of marks corresponding, in order, to the '**LETTER**'s or '**DIGIT**'s contained in that '**TAG**'. These marks may be separated by typographical

display features {9.4.d}. The mark corresponding to each **'LETTER'** (**'DIGIT'**) is the representation of that **LETTER-symbol** (**DIGIT-symbol**). {For example, the representation of a **letter-x-digit-one-symbol** is *x1*, which may be written *x 1*. **TAG-symbols** are used for **identifiers** and **field-selectors**.}

b) The representation, if any, of each **bold-TAG-symbol** is composed of marks corresponding, in order, to the **'LETTER'**s or **'DIGIT'**s contained in that **'TAG'** {but with no typographical display features in between}. The mark corresponding to each **'LETTER'** (**'DIGIT'**) is similar to the mark representing the corresponding **LETTER-symbol** (**DIGIT-symbol**), being, in this Report, the corresponding bold faced letter (digit). {Other methods of indicating the similarity which are recognizable without further elucidation are also acceptable, e.g., ***person***, *person*, *PERSON*, *'person* and *'person'* could all be representations of the **bold-letter-p-letter-e-letter-r-letter-s-letter-o-letter-n-symbol**.}

However, the representation of a **bold-TAG-symbol** may not be the same as any representation of any other **symbol** {; thus there may be a finite number of **bold-TAG-symbols** which have no representation; e.g., there is no representation for the **bold-letter-r-letter-e-letter-a-letter-l-symbol** because ***real*** is a representation of the **real-symbol**; note that the number of **bold-TAG-symbols** available is still arbitrarily large}. If, according to the convention used, a given sequence of marks could be either the representation of one **bold-TAG-symbol** or the concatenation of the representations of two or more other **symbols**, then it is always to be construed as that one **symbol** {; the inclusion of a blank can always force the other interpretation; e.g., ***refreal*** is one **symbol**, whereas ***ref real*** must always be two}. {**Bold-TAG-symbols** are used for **mode-indications** and for **operators**.}

c) The representation of each **SIZE-SIZETY-STANDARD-symbol** is composed of the representation of the corresponding **SIZE-symbol**, possibly followed by typographical display features, followed by the represention of the corresponding **SIZETY-STANDARD-symbol**. {For example, the representation of a **long-real-symbol** is ***long real***, or perhaps *'long''real'* (but not, according to section b above, ***longreal*** or *'longreal'*, for those would be representations of the **bold-letter-l-letter-o-letter-n-letter-g-letter-r-letter-e-letter-a-letter-l-symbol**). **SIZETY-STANDARD-symbols** are used for **mode-indications**.}

d) The representation of each **DOP-cum-becomes-symbol** (**DOP-cum-assigns-to-symbol**) is composed of the mark or marks representing the corresponding **DOP-symbol** followed {without intervening typographical display features} by the marks representing the **becomes-symbol** (the **assigns-to-symbol**). {For example, the representation of a **plus-cum-becomes-symbol** is +:=. **DOP-cum-becomes-symbols** are used for **operators**.}

e) The representation of each **DYAD-cum-NOMAD-symbol** is composed of the mark representing the corresponding **DYAD-symbol** followed {without intervening typographical display features} by the mark representing the corresponding **NOMAD-symbol**. {For example, the representation of an **over-cum-times-symbol** is ÷×. **DYAD-cum-NOMAD-symbols** are used for **operators**, but note that **NOMAD1-cum-NOMAD2-symbols** may be only **dyadic-operators**.}

# PART V

## Environment and Examples

### 10. Standard environment

{The "standard environment" encompasses the constituent **EXTERNAL-preludes**, **system-tasks** and **particular-postludes** of a **program-text**.}

#### 10.1. Program texts

{The programmer is concerned with **particular-programs** (10.1.1.g). These are always included in a **program-text** (10.1.1.a) which also contains the **standard-prelude**, a **library-prelude**, which depends upon the implementation, a **system-prelude** and **system-tasks**, which correspond to the operating environment, possibly some other **particular-programs**, one or more **particular-preludes** (one for each **particular-program**) and one or more **particular-postludes**.}

##### 10.1.1. Syntax

A) **EXTERNAL :: standard ; library ; system ; particular.**
B) **STOP :: label letter s letter t letter o letter p.**

a) **program text : STYLE begin{94f} token, new LAYER1 preludes{b}, parallel{94f} token, new LAYER1 tasks{d} PACK, STYLE end{94f} token.**
b) **NEST1 preludes{a} : NEST1 standard prelude with DECS1{c}, NEST1 library prelude with DECSETY2{c}, NEST1 system prelude with DECSETY3{c}, where (NEST1) is (new EMPTY new DECS1 DECSETY2 DECSETY3).**
c) **NEST1 EXTERNAL prelude with DECSETY1{b,f} : strong void NEST1 series with DECSETY1{32b}, go on{94f} token ; where (DECSETY1) is (EMPTY), EMPTY.**
d) **NEST1 tasks{a} : NEST1 system task{e} list, and also{94f} token, NEST1 user task{f} PACK list.**
e) **NEST1 system task{d} : strong void NEST1 unit{32d}.**

f) **NEST1 user task**{d} : **NEST2 particular prelude with DECS**{c},
**NEST2 particular program**{g} **PACK, go on**{94f} **token,**
**NEST2 particular postlude**{i},
**where (NEST2) is (NEST1 new DECS STOP).**

g) **NEST2 particular program**{f} :
**NEST2 new LABSETY3 joined label definition of LABSETY3**{h},
**strong void NEST2 new LABSETY3**
**ENCLOSED clause**{31a,33a,c,34a,35a}.

h) **NEST joined label definition of LABSETY**{g,h} :
**where (LABSETY) is (EMPTY), EMPTY ;**
**where (LABSETY) is (LAB1 LABSETY1),**
**NEST label definition of LAB1**{32c},
**NEST joined label definition of LABSETY1**{h}.

i) **NEST2 particular postlude**{f} :
**strong void NEST2 series with STOP**{32b}.

{Examples:

a) *(* **c** *standard-prelude* **c**; **c** *library-prelude* **c**; **c** *system-prelude* **c**;
***par begin*** **c** *system-task-1* **c** , **c** *system-task-2* **c** ,
*(* **c** *particular-prelude* **c**;
*(start: commence:* ***begin skip end****);*
**c** *particular-postlude* **c** *)* ,
*(* **c** *another user-task* **c** *)*
***end****)*

b) **c** *standard-prelude* (10.2, 10.3) **c**; **c** *library-prelude* **c**;
**c** *system-prelude* (10.4.1) **c**;

d) **c** *system-task-1* (10.4.2.a) **c** , **c** *system-task-2* **c** ,
*(* **c** *particular-prelude* **c**;
*(start: commence:* ***begin skip end****);*
**c** *particular-postlude* **c** *)* ,
*(* **c** *another user-task* **c** *)*

f) **c** *particular-prelude* (10.5.1) **c**;
*(start: commence:* ***begin skip end****);*
**c** *particular-postlude* (10.5.2) **c**

g) *start: commence:* ***begin skip end***

h) *start: commence:*

i) *stop: lock (stand in); lock (stand out); lock (stand back)* }

10.1.2. The environment condition

a) A **program** in the strict language must be akin {1.1.3.2.k} to some **program-text** whose constituent **EXTERNAL-preludes** and **particular-postludes** are as specified in the remainder of this section.

{It is convenient to speak of the **standard-prelude**, the **library-prelude**, the **particular-programs**, etc. of a **program** when discussing those parts of that **program** which correspond to the constituent **standard-prelude**, etc. of the corresponding **program-text**.}

b) The constituent **standard-prelude** of all **program-texts** is that **standard-prelude** whose representation is obtained {10.1.3} from the forms given in sections 10.2 and 10.3.

c) The constituent **library-prelude** of a **program-text** is not specified in this Report {but must be specified for each implementation; the syntax of '**program text**' ensures that a **declaration** contained in a **library-prelude** may not contradict any **declaration** contained in the **standard-prelude**}.

d) The constituent **system-prelude** (**system-task-list**) of all **program-texts** is that **system-prelude** (**system-task-list**) whose representation is obtained from the forms given in section 10.4, with the possible addition of other forms not specified in this Report {but to be specified to suit the operating environment of each implementation}.

e) Each constituent **particular-prelude** (**particular-postlude**) of all **program-texts** is that **particular-prelude** (**particular-postlude**) whose representation is obtained from the forms given in section 10.5, with the possible addition of other forms not specified in this Report {but to be specified for each implementation}.

10.1.3. The method of description of the standard environment

A representation of an **EXTERNAL-prelude**, **system-task** or **particular-postlude** is obtained by altering each form in the relevant sections of this chapter in the following steps:

Step 1: If a given form F begins with ***op*** {the **operator-symbol**} followed by one of the marks ***P***, ***Q***, ***R*** or ***E***, then F is replaced by a number of new forms each of which is a copy of F in which that mark {following the ***op***} is (all other occurrences in F of that mark are) replaced, in each respective new form, by:

Case A: The mark is ***P***:
- −, +, ×, * or /
(−, +, × or /);

Case B: The mark is ***Q***:
- ***minusab***, −:=, ***plusab***, +:=, ***timesab***, ×:=, *:= or ***divab***, /:=
(−:=, +:=, ×:= or /:=);

Case C: The mark is ***R***:
- <, ***lt***, ≤, <=, ***le***, =, ***eq***, ≠, /=, ***ne***, ≥, >=, ***ge*** or >, ***gt***
(<, ≤, =, ≠, ≥ or >);

Case D: The mark is ***E***:
- =, ***eq*** or ≠, /=, ***ne***
(= or ≠);

Step 2: If, in some form, as possibly made in the step above, ? occurs followed by an **INDICATOR** (a **field-selector**) **I**, then that occurrence of

? is deleted and each **INDICATOR (field-selector)** akin {1.1.3.2.k} to **I** contained in any form is replaced by a copy of one same **INDICATOR (field-selector)** which does not occur elsewhere in the **program** and Step 2 is taken again;

Step 3: If a given form F, as possibly modified or made in the steps above, begins with ***op*** {the **operator-symbol**} followed by a chain of **TAO-symbols** separated by **and-also-symbols**, the chain being enclosed between ⫞ and ⫟, then F is replaced by a number of different "versions" of that form each of which is a copy of F in which that chain, together with its enclosing ⫞ and ⫟, has been replaced by one of those **TAO-symbols** {; however, an implementation is not obliged to provide more than one such version (9.4.b)};

Step 4: If, in a given form, as possibly modified or made in the steps above, there occurs a sequence S of **symbols** enclosed between ⫞ and ⫟ and if, in that S, ***L int***, ***L real***, ***L compl***, ***L bits*** or ***L bytes*** occurs, then S is replaced by a chain of a sufficient number of sequences separated by **and-also-symbols**, the n-th of which is a copy of S in which copy each occurrence of *L* (***L***, ***K***, ***S***) is replaced by (n − 1) times *long* (***long***, ***leng***, ***shorten***), followed by an **and-also-symbol** and a further chain of a sufficient number of sequences separated by **and-also-symbols**, the m-th of which is a copy of S in which copy each occurrence of *L* (***L***, ***K***, ***S***) has been replaced by m times *short* (***short shorten***, ***leng***); the ⫞ and ⫟ enclosing that S are then deleted;

Step 5: If, in a given form F, as possibly modified or made in the steps above, ***L int*** (***L real***, ***L compl***, ***L bits***, ***L bytes***) occurs, then F is replaced by a sequence of a sufficient number of new forms, the n-th of which is a copy of F in which copy each occurrence of *L* (***L***, ***K***, ***S***) is replaced by (n − 1) times *long* (***long***, ***leng***, ***shorten***), and each occurrence of ***long L*** (*long L*) by n times ***long*** (*long*), followed by a further sequence of a sufficient number of new forms, the m-th of which is a copy of F in which copy each occurrence of *L* (***L***, ***K***, ***S***) is replaced by m times *short* (***short***, ***shorten***, ***leng***), and each occurrence of ***long L*** (*long L*) by (m − 1) times ***short*** (*short*);

Step 6: Each occurrence of *F* (***PRIM***) in any form, as possibly modified or made in the steps above, is replaced by a representation of a **letter-aleph-symbol (primal-symbol)** {9.4.a};

Step 7: If a sequence of representations beginning with and ending with ***c*** occurs in any form, as possibly modified or made in the steps above, then this sequence, which is termed a "pseudo-comment", is replaced by a representation of a **declarer** or **closed-clause** suggested by the sequence;

Step 8: If, in any form, as possibly modified or made in the steps above, a **routine-text** occurs whose calling involves the manipulation of real numbers, then this **routine-text** may be replaced by any other **routine-text** whose calling has approximately the same effect {; the degree of approximation is left undefined in this Report (see also 2.1.3.1.e)};

Step 9: In the case of an **EXTERNAL-prelude**, a form consisting of a **skip-symbol** followed by a **go-on-symbol** {***skip;***} is added at the end.

{The term "sufficient number", as used in Steps 4 and 5 above, implies that no intended **particular-program** should have a different meaning or fail to be produced by the syntax solely on account of an insufficiency of that number.}

Wherever {in the transput declarations} the representation $_{10}$ (\, ⊥) occurs within a **character-denotation** or **string-denotation**, it is to be interpreted as the representation of the **string-item** {8.1.4.1.b} used to indicate "times ten to the power" (an alternative form {, if any,} of "times ten to the power", "plus i times") on external media. {Clearly, these representations have been chosen because of their similarity to those of the **times-ten-to-the-power-symbol** (9.4.1.b) and the **plus-i-times-symbol** (9.4.1.c), but, on media on which these characters are not available, other **string-items** must be chosen (and the **letter-e-symbol** and the **letter-i-symbol** are obvious candidates).}

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

## 10.2. The standard prelude

{The **declarations** of the **standard-prelude** comprise "environment enquiries", which supply information concerning a specific property of the implementation (2.2.2.c), "standard modes", "standard operators and functions", "synchronization operations" and "transput declarations" (which are given in section 10.3).}

### 10.2.1. Environment enquiries

a) ***int*** *int lengths* = ***c*** *1 plus the number of extra lengths of integers* {2.1.3.1.d} ***c*** ;

b) ***int*** *int shorths* = ***c*** *1 plus the number of extra shorths of integers* {2.1.3.1.d} ***c*** ;

c) ***L int*** *L max int* = ***c*** *the largest L integral value* {2.2.2.b} ***c*** ;

d) ***int*** *real lengths* = ***c*** *1 plus the number of extra lengths of real numbers* {2.1.3.1.d} ***c*** ;

e) ***int*** *real shorths* = ***c*** *1 plus the number of extra shorths of real numbers* {2.1.3.1.d} ***c*** ;

f) ***L real*** *L max real* = **c** *the largest L real value* {2.2.2.b} **c** ;

g) ***L real*** *L small real* = **c** *the smallest L real value such that both* ***L*** *1 + L small real* > ***L*** *1 and* ***L*** *1 - L small real* < ***L*** *1* {2.2.2.b} **c** ;

h) ***int*** *bits lengths* = **c** *1 plus the number of extra widths* {j} *of bits* **c** ;

i) ***int*** *bits shorths* = **c** *1 plus the number of extra shorths* {j} *of bits* **c** ;

j) ***int*** *L bits width* = **c** *the number of elements in L bits; see* ***L bits*** {10.2.2.g}; *this number increases (decreases) with the "size", i.e., the number of 'long's (minus the number of 'short's) of which 'L' is composed, until a certain size is reached, viz., "the number of extra widths" (minus "the number of extra shorths") of bits, after which it is constant* **c** ;

k) ***int*** *bytes lengths* = **c** *1 plus the number of extra widths* {m} *of bytes* **c** ;

l) ***int*** *bytes shorths* = **c** *1 plus the number of extra shorths* {m} *of bytes* **c** ;

m) ***int*** *L bytes width* = **c** *the number of elements in L bytes; see* ***L bytes*** {10.2.2.h}; *this number increases (decreases) with the "size", i.e., the number of 'long's (minus the number of 'short's) of which 'L' is composed, until a certain size is reached, viz., "the number of extra widths" (minus "the number of extra shorths") of bytes, after which it is constant* **c** ;

n) ***op abs*** = (***char*** *a*) ***int*** : **c** *the integral equivalent* {2.1.3.1.g} *of the character 'a'* **c** ;

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

p) ***int*** *max abs char* = **c** *the largest integral equivalent* {2.1.3.1.g} *of a character* **c** ;

q) ***char*** *null character* = **c** *some character* **c** ;

r) ***char*** *flip* = **c** *the character used to represent 'true' during transput* {10.3.3.1.a, 10.3.3.2.a} **c** ;

s) ***char*** *flop* = **c** *the character used to represent 'false' during transput* **c** ;

t) ***char*** *errorchar* = **c** *the character used to represent unconvertible arithmetic values* {10.3.2.1.b,c,d,e,f} *during transput* **c** ;

u) ***char*** *blank* = "␣" ;

10.2.2. Standard modes

a) ***mode void*** = **c** *an actual-declarer specifying the mode 'void'* **c** ;

b) ***mode bool*** = **c** *an actual-declarer specifying the mode 'boolean'* **c** ;

c) ***mode L int*** = **c** *an actual-declarer specifying the mode 'L integral'* **c** ;

d) ***mode L real*** = ***c*** *an actual-declarer specifying the mode 'L real'* ***c*** ;

e) ***mode char*** = ***c*** *an actual-declarer specifying the mode 'character'* ***c*** ;

f) ***mode L compl*** = ***struct*** *(****L real*** *re, im)* ;

g) ***mode L bits*** = ***struct*** *(*[ *1* : *L bits width* ] ***bool*** *L F)* ; {See 10.2.1.j}
{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.}

h) ***mode L bytes*** = ***struct*** *(*[ *1* : *L bytes width* ] ***char*** *L F)* ; {See 10.2.1.m}

i) ***mode string*** = ***flex*** [ *1* : *0* ] ***char*** ;

10.2.3. Standard operators and functions

10.2.3.0. Standard priorities

a) ***prio minusab*** = *1*, ***plusab*** = *1*, ***timesab*** = *1*, ***divab*** = *1*, ***overab*** = *1*,
***modab*** = *1*, ***plusto*** = *1*,
−:= = *1*, +:= = *1*, ×:= = *1*, ∗:= = *1*, /:= = *1*, ÷:= = *1*, %:= = *1*, ÷×:= = *1*,
÷∗:= = *1*, %×:= = *1*, %∗:= = *1*, +=: = *1*,

∨ = *2*, ***or*** = *2*,

∧ = *3*, & = *3*, ***and*** = *3*,

= = *4*, ***eq*** = *4*, ≠ = *4*, /= = *4*, ***ne*** = *4*,

< = *5*, ***lt*** = *5*, ≤ = *5*, <= = *5*, ***le*** = *5*, ≥ = *5*, >= = *5*, ***ge*** = *5*, > = *5*, ***gt*** = *5*,

− = *6*, + = *6*,

× = *7*, ∗ = *7*, / = *7*, ÷ = *7*, % = *7*, ***over*** = *7*,
÷× = *7*, ÷∗ = *7*, %× = *7*, %∗ = *7*, ***mod*** = *7*,
□ = *7*, ***elem*** = *7*,

↑ = *8*, ∗∗ = *8*, ↓ = *8*, ***up*** = *8*, ***down*** = *8*, ***shl*** = *8*, ***shr*** = *8*,
***lwb*** = *8*, ***upb*** = *8*, ⌊ = *8*, ⌈ = *8*,

⊥ = *9*, +× = *9*, +∗ = *9*, ***i*** = *9* ;

10.2.3.1. Rows and associated operations

a) ***mode*** ? ***rows*** = ***c*** *an actual-declarer specifying a mode united from* {2.1.3.6.a} *a sufficient set of modes each of which begins with 'row'* ***c*** ;

b) ***op*** ⌊***lwb***, ⌊⌋ = (***int*** *n*, ***rows*** *a)* ***int*** : ***c*** *the lower bound in the n-th bound pair of the descriptor of the value of 'a', if that bound pair exists* ***c*** ;

c) **op** ⦃**upb**, ⌜⦄ = (**int** *n*, **rows** *a*) **int** : **c** *the upper bound in the n-th bound pair of the descriptor of the value of 'a', if that bound pair exists* **c** ;

d) **op** ⦃**lwb**, ⌞⦄ = (**rows** *a*) **int** : *1* ⌞ *a* ;

e) **op** ⦃**upb**, ⌜⦄ = (**rows** *a*) **int** : *1* ⌜ *a* ;

{The term "sufficient set", as used in a above and also in 10.3.2.2.b and d, implies that no intended **particular-program** should fail to be produced (nor any unintended **particular-program** be produced) by the syntax solely on account of an insufficiency of modes in that set.}

10.2.3.2. Operations on boolean operands

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

b) **op** ⦃∧, &, **and**⦄ = (**bool** *a*, *b*) **bool** : *(a* | *b* | **false***)* ;

c) **op** ⦃¬, ~, **not**⦄ = (**bool** *a*) **bool** : *(a* | **false** | **true***)* ;

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

e) **op** ⦃≠, /=, **ne**⦄ = (**bool** *a*, *b*) **bool** : ¬ *(a = b)* ;

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

10.2.3.3. Operations on integral operands

a) **op** ⦃<, **lt**⦄ = (**L int** *a*, *b*) **bool** : **c** *true if the value of 'a' is smaller than* {2.1.3.1.e} *that of 'b' and false otherwise* **c** ;

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

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

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

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

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

g) **op** − = (**L int** *a*, *b*) **L int** : **c** *the value of 'a' minus* {2.1.3.1.e} *that of 'b'* **c** ;

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

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

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

k) **op abs** = (**L int** *a*) **L int** : *(a* < **L** *0* | −*a* | *a)* ;

l)
```
op ⦃×, *⦄ = (L int a, b) L int :
  begin L int s := L 0, i := abs b;
    while i ≥ L 1
    do s := s + a; i := i − L 1 od;
    (b < L 0 | − s | s)
  end ;
```

m) **op** {÷, %, **over**} = (**L int** *a*, *b*) **L int**:
**if** *b* ≠ **L** *0*
**then L int** *q* := **L** *0*, *r* := **abs** *a*;
**while** (*r* := *r* − **abs** *b*) ≥ **L** *0* **do** *q* := *q* + **L** *1* **od**;
(*a* < **L** *0* ∧ *b* ≥ **L** *0* ∨ *a* ≥ **L** *0* ∧ *b* < **L** *0* | − *q* | *q*)
**fi**;

n) **op** {÷×, ÷*, %×, %*, **mod**} = (**L int** *a*, *b*) **L int**:
(**int** *r* = *a* − *a* ÷ *b* × *b*; *r* < *0* | *r* + **abs** *b* | *r*);

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

p) **op** {↑, **, **up**} = (**L int** *a*, **int** *b*) **L int**:
(*b* ≥ *0* | **L int** *p* := **L** *1*; **to** *b* **do** *p* := *p* × *a* **od**; *p*);

q) **op leng** = (**L int** *a*) **long L int**: **c** *the long L integral value lengthened from* {2.1.3.1.e} *the value of 'a'* **c**;

r) **op shorten** = (**long L int** *a*) **L int**: **c** *the L integral value, if it exists, which can be lengthened to* {2.1.3.1.e} *the value of 'a'* **c**;

s) **op odd** = (**L int** *a*) **bool**: **abs** *a* ÷× **L** *2* = **L** *1*;

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

u) **op** {⊥, +×, +*, *i*} = (**L int** *a*, *b*) **L compl**: (*a*, *b*);

10.2.3.4. Operations on real operands

**a)** **op** {<, **lt**} = (**L real** *a*, *b*) **bool**: **c** *true if the value of 'a' is smaller than* {2.1.3.1.e} *that of 'b' and false otherwise* **c**;

b) **op** {≤, <=, **le**} = (**L real** *a*, *b*) **bool**: ¬ (*b* < *a*);

c) **op** {=, **eq**} = (**L real** *a*, *b*) **bool**: *a* ≤ *b* ∧ *b* ≤ *a*;

d) **op** {≠, /=, **ne**} = (**L real** *a*, *b*) **bool**: ¬ (*a* = *b*);

e) **op** {≥, >=, **ge**} = (**L real** *a*, *b*) **bool**: *b* ≤ *a*;

f) **op** {>, **gt**} = (**L real** *a*, *b*) **bool**: *b* < *a*;

g) **op** − = (**L real** *a*, *b*) **L real**: **c** *the value of 'a' minus* {2.1.3.1.e} *that of 'b'* **c**;

h) **op** − = (**L real** *a*) **L real**: **L** *0* − *a*;

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

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

k) **op abs** = (**L real** *a*) **L real**: (*a* < **L** *0* | − *a* | *a*);

l) **op** {×, *} = (**L real** *a*, *b*) **L real**: **c** *the value of 'a' times* {2.1.3.1.e} *that of 'b'* **c**;

m) **op** / = (**L real** *a*, *b*) **L real** : **c** *the value of 'a' divided by* {2.1.3.1.e} *that of 'b'* **c** ;

n) **op leng** = (**L real** *a*) **long L real** : **c** *the long L real value lengthened from* {2.1.3.1.e} *the value of 'a'* **c** ;

o) **op shorten** = (**long L real** *a*) **L real** : **c** *if* **abs** *a* ≤ **leng** *L max real, then a L real value 'v' such that, for any L real value 'w',* **abs** (**leng** *v* − *a*) ≤ **abs** (**leng** *w* − *a*) **c** ;

p) **op round** = (**L real** *a*) **L int** : **c** *a L integral value, if one exists, which is widenable to* {2.1.3.1.e} *a L real value differing by not more than one-half from the value of 'a'* **c** ;

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

r) **op** ⌊**entier**, ᵗ⌋ = (**L real** *a*) **L int** :
**begin L int** *j* := **L** *0*;
**while** *j* < *a* **do** *j* := *j* + **L** *1* **od**;
**while** *j* > *a* **do** *j* := *j* − **L** *1* **od**;
*j*
**end** ;

s) **op** ⌊⊥, +×, +*, *i*⌋ = (**L real** *a*, *b*) **L compl** : (*a*, *b*) ;

10.2.3.5. Operations on arithmetic operands

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

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

c) **op** ***R*** = (**L real** *a*, **L int** *b*) **bool** : *a* ***R*** **L real** (*b*) ;

d) **op** ***R*** = (**L int** *a*, **L real** *b*) **bool** : **L real** (*a*) ***R*** *b* ;

e) **op** ⌊⊥, +×, +*, *i*⌋ = (**L real** *a*, **L int** *b*) **L compl** : (*a*, *b*) ;

f) **op** ⌊⊥, +×, +*, *i*⌋ = (**L int** *a*, **L real** *b*) **L compl** : (*a*, *b*) ;

g) **op** ⌊↑, **, **up**⌋ = (**L real** *a*, **int** *b*) **L real** :
(**L real** *p* := **L** *1*; **to abs** *b* **do** *p* := *p* × *a* **od**; (*b* ≥ *0* | *p* | **L** *1* / *p*)) ;

10.2.3.6. Operations on character operands

a) **op** ***R*** = (**char** *a*, *b*) **bool** : **abs** *a* ***R*** **abs** *b* ; {10.2.1.n}

b) **op** + = (**char** *a*, *b*) **string** : (*a*, *b*) ;

10.2.3.7. Operations on complex operands

a) **op re** = (**L compl** *a*) **L real** : *re* **of** *a* ;

b) **op im** = (**L compl** *a*) **L real** : *im* **of** *a* ;

c) **op abs** = (**L compl** *a*) **L real** : *L sqrt* (**re** *a* ↑ 2 + **im** *a* ↑ 2) ;

d) **op arg** = (**L compl** *a*) **L real** :
**if L real** *re* = **re** *a*, *im* = **im** *a*;
*re* ≠ **L** 0 ∨ *im* ≠ **L** 0
**then if abs** *re* > **abs** *im*
**then** *L arctan* (*im* / *re*) + *L pi* / **L** 2 ×
(*im* < **L** 0 | **sign** *re* − 1 | 1 − **sign** *re*)
**else** −*L arctan* (*re* / *im*) + *L pi* / **L** 2 × **sign** *im*
**fi**
**fi** ;

e) **op conj** = (**L compl** *a*) **L compl** : **re** *a* ⊥ −**im** *a* ;

f) **op** ↑=, **eq**↑ = (**L compl** *a*, *b*) **bool** : **re** *a* = **re** *b* ∧ **im** *a* = **im** *b* ;

g) **op** ↑≠, /=, **ne**↑ = (**L compl** *a*, *b*) **bool** : ¬ (*a* = *b*) ;

h) **op** − = (**L compl** *a*, *b*) **L compl** : (**re** *a* − **re** *b*) ⊥ (**im** *a* − **im** *b*) ;

i) **op** −= (**L compl** *a*) **L compl** : − **re** *a* ⊥ −**im** *a* ;

j) **op** + = (**L compl** *a*, *b*) **L compl** : (**re** *a* + **re** *b*) ⊥ (**im** *a* + **im** *b*) ;

k) **op** + = (**L compl** *a*) **L compl** : *a* ;

l) **op** ↑×, ∗↑ = (**L compl** *a*, *b*) **L compl** :
(**re** *a* × **re** *b* − **im** *a* × **im** *b*) ⊥ (**re** *a* × **im** *b* + **im** *a* × **re** *b*) ;

m) **op** / = (**L compl** *a*, *b*) **L compl** :
(**L real** *d* = **re** (*b* × **conj** *b*); **L compl** *n* = *a* × **conj** *b*;
(**re** *n* / *d*) ⊥ (**im** *n* / *d*)) ;

n) **op leng** = (**L compl** *a*) **long L compl** : **leng re** *a* ⊥ **leng im** *a* ;

o) **op shorten** = (**long L compl** *a*) **L compl** :
**shorten re** *a* ⊥ **shorten im** *a* ;

p) **op P** = (**L compl** *a*, **L int** *b*) **L compl** : *a* **P L compl** (*b*) ;

q) **op P** = (**L compl** *a*, **L real** *b*) **L compl** : *a* **P L compl** (*b*) ;

r) **op P** = (**L int** *a*, **L compl** *b*) **L compl** : **L compl** (*a*) **P** *b* ;

s) **op P** = (**L real** *a*, **L compl** *b*) **L compl** : **L compl** (*a*) **P** *b* ;

t) **op** ↑↑, ∗∗, **up**↑ = (**L compl** *a*, **int** *b*) **L compl** :
(**L compl** *p* := **L** 1; **to abs** *b* **do** *p* := *p* × *a* **od**; (*b* ≥ 0 | *p* | **L** 1 / *p*)) ;

u) **op E** = (**L compl** *a*, **L int** *b*) **bool** : *a* **E L compl** (*b*) ;

v) **op E** = (**L compl** *a*, **L real** *b*) **bool** : *a* **E L compl** (*b*) ;

w) **op E** = (**L int** *a*, **L compl** *b*) **bool** : *b* **E** *a* ;

x) **op E** = (**L real** *a*, **L compl** *b*) **bool** : *b* **E** *a* ;

10.2.3.8. Bits and associated operations

a)
```
op ≠, eq ≠ = (L bits a, b) bool :
  begin bool c;
    for i to L bits width
    while c := (L F of a) [i] = (L F of b) [i]
    do skip od;
    c
  end ;
```

b)
```
op ≠ ≠, /=, ne ≠ = (L bits a, b) bool : ¬ (a = b) ;
```

c)
```
op ≠ ∨, or ≠ = (L bits a, b) L bits :
  begin L bits c;
    for i to L bits width
    do (L F of c) [i] := (L F of a) [i] ∨ (L F of b) [i] od;
    c
  end ;
```

d)
```
op ≠ ∧, &, and ≠ = (L bits a, b) L bits :
  begin L bits c;
    for i to L bits width
    do (L F of c) [i] := (L F of a) [i] ∧ (L F of b) [i] od;
    c
  end ;
```

e)
```
op ≠ ≤, <=, le ≠ = (L bits a, b) bool : (a ∨ b) = b ;
```

f)
```
op ≠ ≥, >=, ge ≠ = (L bits a, b) bool : b ≤ a ;
```

g)
```
op ≠ ↑, up, shl ≠ = (L bits a, int b) L bits :
  if abs b ≤ L bits width
  then L bits c := a;
    to abs b
    do if b > 0 then
      for i from 2 to L bits width
      do (L F of c) [i - 1] := (L F of c) [i] od;
      (L F of c) [L bits width] := false
    else
      for i from L bits width by -1 to 2
      do (L F of c) [i] := (L F of c) [i - 1] od;
      (L F of c) [1] := false
    fi od;
    c
  fi ;
```

h)
```
op ≠ ↓, down, shr ≠ = (L bits x, int n) L bits : x ↑ - n ;
```

```
i)   op abs = (L bits a) L int :
       begin L int c := L 0;
           for i to L bits width
           do c := L 2 × c + K abs (L F of a) [i] od;
           c
       end ;

j)   op bin = (L int a) L bits :
       if a ≥ L 0
       then L int b := a; L bits c;
           for i from L bits width by - 1 to 1
           do (L F of c) [i] := odd b; b := b ÷ L 2 od;
           c
       fi ;

k)   op ↑elem, □↑ = (int a, L bits b) bool : (L F of b) [a] ;

l)   proc L bits pack = ([ ] bool a) L bits :
       if int n = ↑ a [@ 1];
           n ≤ L bits width
       then L bits c;
           for i to L bits width
           do (L F of c) [i] :=
               (i ≤ L bits width - n | false | a [@ 1] [i - L bits width + n])
           od;
           c
       fi ;

m)   op ↑¬, ~, not↑ = (L bits a) L bits :
       begin L bits c;
           for i to L bits width do (L F of c) [i] := ¬ (L F of a) [i] od;
           c
       end ;

n)   op leng = (L bits a) long L bits : long L bits pack (a) ;

o)   op shorten = (long L bits a) L bits : L bits pack ([ ] bool (a)
       [long L bits width - L bits width + 1 : ]);
```

10.2.3.9. Bytes and associated operations

```
a)   op R = (L bytes a, b) bool : string (a) R string (b) ;

b)   op ↑elem, □↑ = (int a, L bytes b) char : (L F of b) [a] ;

c)   proc L bytes pack = (string a) L bytes :
       if int n = ↑ a [@ 1];
           n ≤ L bytes width
       then L bytes c;
           for i to L bytes width
           do (L F of c) [i] := (i ≤ n | a [@ 1] [i] | null character) od;
           c
       fi ;
```

d) **op leng** = (**L bytes** *a*) **long L bytes** : *long L bytes pack (a)* ;

e) **op shorten** = (**long L bytes** *a*) **L bytes** :
*L bytes pack* (**string** *(a)* [ : *L bytes width* ]) ;

10.2.3.10. Strings and associated operations

a) **op** ⦃<, **lt**⦄ = (**string** *a, b*) **bool** :
**begin int** *m* = ⌜ *a* [@ *1* ], *n* = ⌜ *b* [@ *1* ]; **int** *c* := *0*;
**for** *i* **to** (*m* < *n* | *m* | *n*)
**while** (*c* := **abs** *a* [@ *1* ] [*i*] − **abs** *b* [@ *1* ] [*i*]) = *0*
**do skip od**;
(*c* = *0* | *m* < *n* ∧ *n* > *0* | *c* < *0*)
**end** ;

b) **op** ⦃≤, <=, **le**⦄ = (**string** *a, b*) **bool** : ¬ (*b* < *a*) ;

c) **op** ⦃=, **eq**⦄ = (**string** *a, b*) **bool** : *a* ≤ *b* ∧ *b* ≤ *a* ;

d) **op** ⦃≠, /=, **ne**⦄ = (**string** *a, b*) **bool** : ¬ (*a* = *b*) ;

e) **op** ⦃≥, >=, **ge**⦄ = (**string** *a, b*) **bool** : *b* ≤ *a* ;

f) **op** ⦃>, **gt**⦄ = (**string** *a, b*) **bool** : *b* < *a* ;

g) **op R** = (**string** *a*, **char** *b*) **bool** : *a* **R string** *(b)* ;

h) **op R** = (**char** *a*, **string** *b*) **bool** : **string** *(a)* **R** *b* ;

i) **op** + = (**string** *a, b*) **string** :
(**int** *m* = (**int** *la* = ⌜ *a* [@ *1* ]; *la* < *0* | *0* | *la*),
*n* = (**int** *lb* = ⌜ *b* [@ *1* ]; *lb* < *0* | *0* | *lb*);
[ *1* : *m* + *n* ] **char** *c*;
*c* [ *1* : *m* ] := *a* [@ *1* ]; *c* [ *m* + *1* : *m* + *n* ] := *b* [@ *1* ]; *c*) ;

j) **op** + = (**string** *a*, **char** *b*) **string** : *a* + **string** *(b)* ;

k) **op** + = (**char** *a*, **string** *b*) **string** : **string** *(a)* + *b* ;

l) **op** ⦃×, *⦄ = (**string** *a*, **int** *b*) **string** : (**string** *c*; **to** *b* **do** *c* := *c* + *a* **od**; *c*) ;

m) **op** ⦃×, *⦄ = (**int** *a*, **string** *b*) **string** : *b* × *a* ;

n) **op** ⦃×, *⦄ = (**char** *a*, **int** *b*) **string** : **string** *(a)* × *b* ;

o) **op** ⦃×, *⦄ = (**int** *a*, **char** *b*) **string** : *b* × *a* ;

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

10.2.3.11. Operations combined with assignations

a) **op** ⦃**minusab**, −:=⦄ = (**ref L int** *a*, **L int** *b*) **ref L int** : *a* := *a* − *b* ;

b) **op** ⦃**minusab**, −:=⦄ = (**ref L real** *a*, **L real** *b*) **ref L real** : *a* := *a* − *b* ;

c) **op** ⦃**minusab**, −:=⦄ = (**ref L compl** *a*, **L compl** *b*) **ref L compl** :
*a* := *a* − *b* ;

d) **op** ¢**plusab**, +:=¢ = (**ref L int** *a*, **L int** *b*) **ref L int** : *a* := *a* + *b* ;

e) **op** ¢**plusab**, +:=¢ = (**ref L real** *a*, **L real** *b*) **ref L real** : *a* := *a* + *b* ;

f) **op** ¢**plusab**, +:=¢ = (**ref L compl** *a*, **L compl** *b*) **ref L compl** : *a* := *a* + *b* ;

g) **op** ¢**timesab**, ×:=, *:=¢ = (**ref L int** *a*, **L int** *b*) **ref L int** : *a* := *a* × *b* ;

h) **op** ¢**timesab**, ×:=, *:=¢ = (**ref L real** *a*, **L real** *b*) **ref L real** : *a* := *a* × *b* ;

i) **op** ¢**timesab**, ×:=, *:=¢ = (**ref L compl** *a*, **L compl** *b*) **ref L compl** :
*a* := *a* × *b* ;

j) **op** ¢**overab**, ÷:=, %:=¢ = (**ref L int** *a*, **L int** *b*) **ref L int** : *a* := *a* ÷ *b* ;

k) **op** ¢**modab**, ÷×:=, ÷*:=, %×:=, %*:=¢ =
(**ref L int** *a*, **L int** *b*) **ref L int** : *a* := *a* ÷× *b* ;

l) **op** ¢**divab**, /:=¢ = (**ref L real** *a*, **L real** *b*) **ref L real** : *a* := *a* / *b* ;

m) **op** ¢**divab**, /:=¢ = (**ref L compl** *a*, **L compl** *b*) **ref L compl** : *a* := *a* / *b* ;

n) **op Q** = (**ref L real** *a*, **L int** *b*) **ref L real** : *a* **Q L real** (*b*) ;

o) **op Q** = (**ref L compl** *a*, **L int** *b*) **ref L compl** : *a* **Q L compl** (*b*) ;

p) **op Q** = (**ref L compl** *a*, **L real** *b*) **ref L compl** : *a* **Q L compl** (*b*) ;

q) **op** ¢**plusab**, +:=¢ = (**ref string** *a*, **string** *b*) **ref string** : *a* := *a* + *b* ;

r) **op** ¢**plusto**, +=:¢ = (**string** *a*, **ref string** *b*) **ref string** : *b* := *a* + *b* ;

s) **op** ¢**plusab**, +:=¢ = (**ref string** *a*, **char** *b*) **ref string** : *a* +:= **string** (*b*) ;

t) **op** ¢**plusto**, +=:¢ = (**char** *a*, **ref string** *b*) **ref string** : **string** (*a*) +=: *b* ;

u) **op** ¢**timesab**, ×:=, *:=¢ = (**ref string** *a*, **int** *b*) **ref string** : *a* := *a* × *b* ;

10.2.3.12. Standard mathematical constants and functions

a) **L real** *L pi* = **c** *a L real value close to* π ; *see Math. of Comp. v. 16, 1962, pp. 80–99* **c** ;

b) **proc** *L sqrt* = (**L real** *x*) **L real** : **c** *if x* ≥ **L** *0, a L real value close to the square root of 'x'* **c** ;

c) **proc** *L exp* = (**L real** *x*) **L real** : **c** *a L real value, if one exists, close to the exponential function of 'x'* **c** ;

d) **proc** *L ln* = (**L real** *x*) **L real** : **c** *a L real value, if one exists, close to the natural logarithm of 'x'* **c** ;

e) **proc** *L cos* = (**L real** *x*) **L real** : **c** *a L real value close to the cosine of 'x'* **c** ;

f) **proc** *L arccos* = (**L real** *x*) **L real** : **c** *if* **abs** *x* ≤ **L** *1, a L real value close to the inverse cosine of 'x',* **L** *0* ≤ *L arccos (x)* ≤ *L pi* **c** ;

g) **proc** *L sin* = (**L real** *x*) **L real** : **c** *a L real value close to the sine of 'x'* **c** ;

h) ***proc*** *L arcsin = (**L real** x)* ***L real*** : ***c*** *if* ***abs*** *x ≤* ***L*** *1, a L real value close to the inverse sine of 'x',* ***abs*** *L arcsin (x) ≤ L pi / **L** 2* ***c*** ;

i) ***proc*** *L tan = (**L real** x)* ***L real*** : ***c*** *a L real value, if one exists, close to the tangent of 'x'* ***c*** ;

j) ***proc*** *L arctan = (**L real** x)* ***L real*** : ***c*** *a L real value close to the inverse tangent of 'x',* ***abs*** *L arctan (x) ≤ L pi / **L** 2* ***c*** ;

k) ***proc*** *L next random = (**ref L int** a)* ***L real*** :
*(a :=* ***c*** *the next pseudo-random L integral value after 'a' from a uniformly distributed sequence on the interval [**L** 0, L max int]* ***c*** ;
***c*** *the real value corresponding to 'a' according to some mapping of integral values [**L** 0, L max int] into real values [**L** 0, **L** 1)* {i.e., such that 0 ≤ x < 1} *such that the sequence of real values so produced preserves the properties of pseudo-randomness and uniform distribution of the sequence of integral values* ***c***) ;

10.2.4. Synchronization operations

The elaboration of a **parallel-clause** P {3.3.1.c} in an environ E is termed a "parallel action". The elaboration of a constituent **unit** of P in E is termed a "process" of that parallel action.

Any elaboration A {in some environ} of either of the **ENCLOSED-clauses** delineated by the **pragmats** {9.2.1.b} ***pr*** *start of incompatible part* ***pr*** and ***pr*** *finish of incompatible part* ***pr*** in the forms 10.2.4.d and 10.2.4.e is incompatible with {2.1.4.2.e} any elaboration B of either of those **ENCLOSED-clauses** if A and B are descendent actions {2.1.4.2.b} of different processes of some same parallel action.

a) ***mode sema = struct (ref int F)*** ;

b) ***op level = (int a) sema*** : ***(sema*** *s; F* ***of*** *s :=* ***heap int*** *:= a; s)* ;

c) ***op level = (sema*** *a)* ***int*** : *F* ***of*** *a* ;

d)
```
op down = (sema edsger) void :
  begin ref int dijkstra = F of edsger;
    while
      pr start of incompatible part pr
        if dijkstra ≥ 1 then dijkstra -:= 1; false
        else
          c let P be the process such that the elaboration of
            this pseudo-comment {10.1.3.Step 7} is a descendent
            action of P, but not of any other process descended
            from P; the process P is halted {2.1.4.3.f} c;
          true
        fi
      pr finish of incompatible part pr
    do skip od
  end ;
```

```
e)  op up = (sema edsger) void :
      pr start of incompatible part pr
        if ref int dijkstra = F of edsger; (dijkstra +:= 1) ≥ 1
        then
          c all processes are resumed {2.1.4.3.g} which are halted
          because the integer referred to by the name yielded by
          'dijkstra' was smaller than one c
        fi
      pr finish of incompatible part pr ;
```

{For the use of ***down*** and ***up***, see E.W. Dijkstra, Cooperating Sequential Processes, contained in Programming Languages, Genuys, F. (ed.), London etc., Academic Press, 1968; see also 11.12.}

## 10.3. 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.}

{Three ways of "transput" (i.e., input and output) are provided by the **standard-prelude**, viz., formatless transput (10.3.3), formatted transput (10.3.5) and binary transput (10.3.6).}

### 10.3.1. Books, channels and files

{"Books", "channels" and "files" model the transput devices of the physical machine used in the implementation.}

#### 10.3.1.1. Books and backfiles

{aa) All information within the system is to be found in a number of "books". A book (a) is a structured value including a field *text* of the mode specified by ***flextext*** (b) which refers to information in the form of characters. The *text* has a variable number of pages, each of which may have a variable number of lines, each of which may have a variable number of characters. Positions within the *text* are indicated by a page number, a line number and a character number. The book includes a field *lpos* which indicates the "logical end" of the book, i.e., the position up to which it has been filled with information, a string *idf*, which identifies the book and which may possibly include other information, e.g., ownership, and fields *putting* and *users* which permit the book to be opened (10.3.1.4.d) on more than one file simultaneously only if putting is not possible on any of them.

bb) The books in the system are accessed via a chain of backfiles. The chain of books available for opening (10.3.1.4.dd) is referenced by *chainbfile*. A given book may be referenced by more than one backfile on this chain, thus allowing simultaneous access to a single book by more than one process (10.2.4). However such access can only be for reading a book, since only one process may access a book such that it may be written to (aa). The chain of books which have been locked (10.3.1.4.o) is referenced by *lockedbfile*.

cc) Simultaneous access by more than one process to the chain of backfiles is prevented by use of the semaphore *bfileprotect*, which provides mutual exclusion between such processes.

dd) Books may be created (e.g., by input) or destroyed (e.g., after output) by tasks (e.g., the operating system) in the **system-task-list** (10.4.2), such books being then added to or removed from the chain of backfiles.)

a) ***mode*** ? ***book*** =
***struct*** *(* ***flextext*** *text,*
***pos*** *lpos* ¢ *logical end of book* ¢,
***string*** *idf* ¢ *identification* ¢,
***bool*** *putting* ¢ *true if the book may be written to* ¢,
***int*** *users* ¢ *the number of times the book is opened* ¢*)* ;

b) ***mode*** ? ***text*** = ***ref*** [ ] [ ] [ ] ***char***,
***mode*** ? ***flextext*** = ***ref flex*** [ ] ***flex*** [ ] ***flex*** [ ] ***char*** ;

c) ***mode*** ? ***pos*** = ***struct*** *(* ***int*** *p, l, c)* ;

d) ***prio*** ? ***beyond*** = 5,
***op beyond*** = *(* ***pos*** *a, b)* ***bool*** :
***if*** *p* ***of*** *a* < *p* ***of*** *b* ***then false***
***elif*** *p* ***of*** *a* > *p* ***of*** *b* ***then true***
***elif*** *l* ***of*** *a* < *l* ***of*** *b* ***then false***
***elif*** *l* ***of*** *a* > *l* ***of*** *b* ***then true***
***else*** *c* ***of*** *a* > *c* ***of*** *b*
***fi*** ;

e) ***mode*** ? ***bfile*** = ***struct*** *(* ***ref book*** *book,* ***ref bfile*** *next)* ;

f) ***ref bfile*** ? *chainbfile* := ***nil*** ;

g) ***ref bfile*** ? *lockedbfile* := ***nil*** ;

h) ***sema*** ? *bfileprotect* = *(* ***sema*** *s; F* ***of*** *s* := ***PRIM int*** := *1; s)* ;

10.3.1.2. Channels

(aa) A "channel" corresponds to one or more physical devices (e.g., a card reader, a card punch or a line printer, or even to a set up in nuclear

physics the results of which are collected by the computer), or to a filestore maintained by the operating system. A channel is a structured value whose fields are routines returning truth values which determine the available methods of access to a book linked via that channel. Since the methods of access to a book may well depend on the book as well as on the channel (e.g., a certain book may have been trapped so that it may be read, but not written to), most of these properties depend on both the channel and the book. These properties may be examined by use of the environment enquiries provided for files (10.3.1.3.ff). Two environment enquiries are provided for channels. These are:

- *estab possible*, which returns true if another file may be "established" (10.3.1.4.cc) on the channel;
- *standconv*, which may be used to obtain the default "conversion key" (bb) for the channel.

bb) A "conversion key" is a value of the mode specified by **conv** which is used to convert characters to and from the values as stored in "internal" form and as stored in "external" form in a book. It is a structured value comprising a row of structures, each of which contains a value in internal form and its corresponding external value. The implementation may provide additional conversion keys in its **library-prelude**.

cc) Three standard channels are provided, with properties as defined below (e,f,g). The implementation may provide additional channels in its **library-prelude**. The *channel number* field is provided in order that different channels with otherwise identical possibilities may be distinguished.]

a) **mode channel** =
**struct (proc (ref book) bool** ? *reset*, ? *set*, ? *get*, ? *put*, ? *bin*,
? *compress*, ? *reidf*,
**proc bool** ? *estab*, **proc pos** ? *max pos*,
**proc (ref book) conv** ? *standconv*, **int** ? *channel number*) ;

b) **mode** ? **conv** = **struct** ([ 1 : **int (skip)** ] **struct (char** *internal, external*) *F*) ;

c) **proc** *estab possible* = (**channel** *chan*) **bool** : *estab* **of** *chan* ;

d) **proc** *standconv* = (**channel** *chan*) **proc (ref book) conv** :
*standconv* **of** *chan* ;

e) **channel** *stand in channel* = **c** *a channel value whose field selected by 'get' is a routine which always returns true, and whose other fields are some suitable values* **c** ;

f) **channel** *stand out channel* = **c** *a channel value whose field selected by 'put' is a routine which always returns true, and whose other fields are some suitable values* **c** ;

g) ***channel*** *stand back channel* = **c** *a channel value whose fields selected by 'set', 'reset', 'get', 'put' and 'bin' are routines which always return true, and whose other fields are some suitable values* **c** ;

10.3.1.3. Files

(aa) A "file" is the means of communication between a **particular-program** and a book which has been opened on that file via some channel. It is a structured value which includes a reference to the book to which it has been linked (10.3.1.4.bb) and a separate reference to the *text* of the book. The file also contains information necessary for the transput routines to work with the book, including its current position *cpos* in the *text*, its current "state" (bb), its current "format" (10.3.4) and the channel on which it has been opened.

bb) The "state" of a file is determined by five fields:

- *read mood*, which is true if the file is being used for input;
- *write mood*, which is true if the file is being used for output;
- *char mood*, which is true if the file is being used for character transput;
- *bin mood*, which is true if the file is being used for binary transput;
- *opened*, which is true if the file has been linked to a book.

cc) A file includes some "event routines", which are called when certain conditions arise during transput. After opening a file, the event routines provided by default return false when called, but the programmer may provide other event routines. Since the fields of a file are not directly accessible to the user, the event routines may be changed by use of the "on routines" (l,m,n,o,p,q,r). The event routines are always given a reference to the file as a parameter. If the elaboration of an event routine is terminated, then the transput routine which called it can take no further action; otherwise, if it returns true, then it is assumed that the condition has been mended in some way, and, if possible, transput continues, but if it returns false, then the system continues with its default action. The on routines are:

- *on logical file end*. The corresponding event routine is called when, during input from a book or as a result of calling *set*, the logical end of the book is reached (see 10.3.1.6.dd).

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; on logical file end (intape, (ref file file) bool : goto f);
    do get (intape, loc int); n +:= 1 od;
    f: print (n)
end ,
```

then the assignment to the field of *intape* violates the scope restriction, since the scope of the routine *(**ref file** file) **bool**: **goto** f* is smaller than the scope of *intape*, so he has to write:

```
begin int n := 0; file auxin := intape;
    on logical file end (auxin, (ref file file) bool : goto f);
    do get (auxin, loc int); n +:= 1 od;
    f: print (n)
end .
```

• *on physical file end.* The corresponding event routine is called when the current page number of the file exceeds the number of pages in the book and further transput is attempted (see 10.3.1.6.dd).

• *on page end.* The corresponding event routine is called when the current line number exceeds the number of lines in the current page and further transput is attempted (see 10.3.1.6.dd).

• *on line end.* The corresponding event routine is called when the current character number of the file exceeds the number of characters in the current line and further transput is attempted (see 10.3.1.6.dd).

Example:

The programmer wishes automatically to give a heading at the start of each page on his file *f*:

```
on page end (f, proc (ref file file) bool :
    (put (file, (newpage, "page number ", whole (i +:= 1, 0),
        newline)); true)
        ¢ it is assumed that i has been declared elsewhere ¢) .
```

• *on char error.* The corresponding event routine is called when a character conversion was unsuccessful or when, during input, a character is read which was not "expected" (10.3.4.1.ll). The event routine is called with a reference to a character suggested as a replacement. The event routine provided by the programmer may assign some character other than the suggested one. If the event routine returns true, then that suggested character as possibly modified is used.

Example:

The programmer wishes to read sums of money punched as *"$123.45"*, *"␣$23.45"*, *"␣␣$3.45"*, etc.:

```
on char error (stand in, (ref file f, ref char sugg) bool :
    if sugg = "0"
    then char c; backspace (f); get (f, c);
        (c = "$" | get (f, sugg); true | false)
    else false
    fi );
int cents; readf (($ 3z "." dd $, cents))
```

• *on value error*. The corresponding event routine is called when:
(i) during formatted transput an attempt is made to transput a value under the control of a "picture" with which it is incompatible, or when the number of "frames" is insufficient. If the routine returns true, then the current value and picture are skipped and transput continues; if the routine returns false, then first, on output, the value is output by *put*, and next *undefined* is called;
(ii) during input it is impossible to convert a string to a value of some given mode (this would occur if, for example, an attempt were made to read an integer larger than *max int* (10.2.1.c)).

• *on format end*. The corresponding event routine is called when, during formatted transput, the format is exhausted while some value still remains to be transput. If the routine returns true, then *undefined* is called if a new format has not been provided for the file by the routine; otherwise, the current format is repeated.

dd) The *conv* field of a file is its current conversion key (10.3.1.2.bb). After opening a file, a default conversion key is provided. Some other conversion key may be provided by the programmer by means of a **call** of *make conv* (j). Note that such a key must have been provided in the **library-prelude**.

ee) The routine *make term* is used to associate a string with a file. This string is used when inputting a variable number of characters, any of its characters serving as a terminator.

ff) The available methods of access to a book which has been opened on a file may be discovered by **calls** of the following routines (note that the yield of such a **call** may be a function of both the book and the channel, and of other environmental factors not defined by this Report):
• *get possible*, which returns true if the file may be used for input;
• *put possible*, which returns true if the file may be used for output;
• *bin possible*, which returns true if the file may be used for binary transput;
• *compressible*, which returns true if lines and pages will be compressed (10.3.1.6.aa) during output, in which case the book is said to be "compressible";
• *reset possible*, which returns true if the file may be reset, i.e., its current position set to (1, 1, 1);
• *set possible*, which returns true if the file may be set, i.e., the current position changed to some specified value; the book is then said to be a "random access" book and, otherwise, a "sequential access" book;
• *reidf possible*, which returns true if the *idf* field of the book may be changed;
• *chan*, which returns the channel on which the file has been opened (this may be used, for example, by a routine assigned by *on physical file end*, in order to open another file on the same channel).

gg) On sequential access books, *undefined* (10.3.1.4.a) is called if binary and character transput is alternated, i.e., after opening or resetting (10.3.1.6.j), either is possible but, once one has taken place, the other may not until after another reset.

hh) On sequential access books, output immediately causes the logical end of the book to be moved to the current position (unless both are in the same line); thus input may not follow output without first resetting (10.3.1.6.j).

Example:

```
begin file f1, f2; [1 : 10000] int x; int n := 0;
    open (f1, "", channel 2);
    f2 := f1;
        ¢ now f1 and f2 can be used interchangeably ¢
    make conv (f1, flexocode); make conv (f2, telexcode);
        ¢ now f1 and f2 use different codes; flexocode and telexcode are
        defined in the library-prelude for this implementation ¢
    reset (f1);
        ¢ consequently, f2 is reset too ¢
    on logical file end (f1, (ref file f) bool : goto done);
    for i do get (f1, x [i]); n := i od;
        ¢ too bad if there are more than 10000 integers in the input ¢
done:
    reset (f1); for i to n do put (f2, x [i]) od;
    close (f2) ¢ f1 is now closed too ¢
end }
```

a)
```
mode file =
    struct (ref book ? book, union (flextext, text) ? text, channel ? chan,
        ref format ? format, ref int ? forp,
        ref bool ? read mood, ? write mood, ? char mood, ? bin mood,
            ? opened,
        ref pos ? cpos ¢ current position ¢,
        string ? term ¢ string terminator ¢,
        conv ? conv ¢ character conversion key ¢,
        proc (ref file) bool ? logical file mended, ? physical file mended,
            ? page mended, ? line mended, ? format mended,
            ? value error mended,
        proc (ref file, ref char) bool ? char error mended);
```

b)
```
proc get possible = (ref file f) bool :
    (opened of f | (get of chan of f) (book of f) | undefined; skip);
```

c)
```
proc put possible = (ref file f) bool :
    (opened of f | (put of chan of f) (book of f) | undefined; skip);
```

d)
```
proc bin possible = (ref file f) bool :
    (opened of f | (bin of chan of f) (book of f) | undefined; skip);
```

e) **proc** *compressible* = (**ref file** *f*) **bool** :
*(opened* **of** *f* | *(compress* **of** *chan* **of** *f) (book* **of** *f)* | *undefined;* **skip***) ;*

f) **proc** *reset possible* = (**ref file** *f*) **bool** :
*(opened* **of** *f* | *(reset* **of** *chan* **of** *f) (book* **of** *f)* | *undefined;* **skip***) ;*

g) **proc** *set possible* = (**ref file** *f*) **bool** :
*(opened* **of** *f* | *(set* **of** *chan* **of** *f) (book* **of** *f)* | *undefined;* **skip***) ;*

h) **proc** *reidf possible* = (**ref file** *f*) **bool** :
*(opened* **of** *f* | *(reidf* **of** *chan* **of** *f) (book* **of** *f)* | *undefined;* **skip***) ;*

i) **proc** *chan* = (**ref file** *f*) **channel** :
*(opened* **of** *f* | *chan* **of** *f* | *undefined;* **skip***) ;*

j) **proc** *make conv* = (**ref file** *f*, **proc** (**ref book**) **conv** *c*) **void** :
*(opened* **of** *f* | *conv* **of** *f* := *c (book* **of** *f)* | *undefined) ;*

k) **proc** *make term* = (**ref file** *f*, **string** *t*) **void** : *term* **of** *f* := *t ;*

l) **proc** *on logical file end* = (**ref file** *f*, **proc** (**ref file**) **bool** *p*) **void** :
*logical file mended* **of** *f* := *p ;*

m) **proc** *on physical file end* = (**ref file** *f*, **proc** (**ref file**) **bool** *p*) **void** :
*physical file mended* **of** *f* := *p ;*

n) **proc** *on page end* = (**ref file** *f*, **proc** (**ref file**) **bool** *p*) **void** :
*page mended* **of** *f* := *p ;*

o) **proc** *on line end* = (**ref file** *f*, **proc** (**ref file**) **bool** *p*) **void** :
*line mended* **of** *f* := *p ;*

p) **proc** *on format end* = (**ref file** *f*, **proc** (**ref file**) **bool** *p*) **void** :
*format mended* **of** *f* := *p ;*

q) **proc** *on value error* = (**ref file** *f*, **proc** (**ref file**) **bool** *p*) **void** :
*value error mended* **of** *f* := *p ;*

r) **proc** *on char error* = (**ref file** *f*, **proc** (**ref file**, **ref char**) **bool** *p*) **void** :
*char error mended* **of** *f* := *p ;*

s) **proc** *reidf* = (**ref file** *f*, **string** *idf*) **void** :
**if** *opened* **of** *f* ∧ *reidf possible (f)* ∧ *idf ok (idf)*
**then** *idf* **of** *book* **of** *f* := *idf*
**fi** ;

10.3.1.4. Opening and closing files

(aa) When, during transput, something happens which is left undefined, for example by explicitly calling *undefined* (a), this does not imply that the elaboration is catastrophically and immediately interrupted (2.1.4.3.h), but only that some sensible action is taken which is not or cannot be described by this Report alone and is generally implementation-dependent.

bb) A book is "linked" with a file by means of *establish* (b), *create* (c) or *open* (d). The linkage may be terminated by means of *close* (n), *lock* (o) or *scratch* (p).

cc) When a file is "established" on a channel, then a book is generated (5.2.3) with a *text* of the given size, the given identification string, with *putting* set to true, and the logical end of the book at (1, 1, 1). An implementation may require (g) that the characters forming the identification string should be taken from a limited set and that the string should be limited in length. It may also prevent two books from having the same string. If the establishing is completed successfully, then the value 0 is returned; otherwise, some nonzero integer is returned (the value of this integer might indicate why the file was not established successfully).

When a file is "created" on a channel, then a file is established with a book whose *text* has the default size for the channel and whose identification string is undefined.

dd) When a file is "opened", then the chain of backfiles is searched for the first book which is such that *match* (h) returns true. (The precise method of matching is not defined by this Report and will, in general, be implementation dependent. For example, the string supplied as parameter to *open* may include a password of some form.) If the end of the chain of backfiles is reached or if a book has been selected, but *putting* of the book yields true, or if putting to the book via the channel is possible and the book is already open, then the further elaboration is undefined. If the file is already open, an ***up*** *gremlins* provides an opportunity for an appropriate system action on the book previously linked (in case no other copy of the file remains to preserve that linkage).

ee) The routine *associate* may be used to associate a file with a value of the mode specified by either ***ref*** [ ] ***char***, ***ref*** [ ] [ ] ***char*** or ***ref*** [ ] [ ] [ ] ***char***, thus enabling such variables to be used as the book of a file.

ff) When a file is "closed", its book is attached to the chain of backfiles referenced by *chainbfile*. Some **system-task** is then activated by means of an ***up*** *gremlins*. (This may reorganize the chain of backfiles, removing this book, or adding further copies of it. It may also cause the book to be output on some external device.)

gg) When a file is "locked", its book is attached to the chain of backfiles referenced by *lockedbfile*. Some **system-task** is then activated by means of an ***up*** *gremlins*. A book which has been locked cannot be re-opened until some subsequent **system-task** has re-attached the book to the chain of backfiles available for opening.

hh) When a file is "scratched", some **system-task** is activated by means of an ***up*** *gremlins*. (This may cause the book linked to the file to be disposed of in some manner.)}

a)
```
proc ? undefined = int : c some sensible system action yielding an
  integer to indicate what has been done; it is presumed that the
  system action may depend on a knowledge of any values
  accessible {2.1.2.c} inside the locale of any environ which is older
  than that in which this pseudo-comment is being elaborated
  {notwithstanding that no ALGOL 68 construct written here could
  access those values} c ;
```

b)
```
proc establish =
  (ref file file, string idf, channel chan, int p, l, c) int :
  begin
    down bfileprotect;
    PRIM book book :=
      (PRIM flex [1 : p] flex [1 : l] flex [1 : c] char, (1, 1, 1), idf,
        true, 1);
    if file available (chan) ∧ (put of chan) (book)
      ∧ estab of chan ∧ ¬ (pos (p, l, c) beyond max pos of chan)
      ∧ ¬ (pos (1, 1, 1) beyond pos (p, l, c)) ∧ idf ok (idf)
    then
      (opened of file | up gremlins | up bfileprotect);
      file :=
        (book, text of book, chan, skip, skip,
        ¢ state: ¢ heap bool := false, heap bool := true,
          heap bool := false, heap bool := false, heap bool := true,
        heap pos := (1, 1, 1), "", (standconv of chan) (book),
        ¢ event routines: ¢ false, false, false, false, false, false,
          (ref file f, ref char a) bool : false);
      (¬ bin possible (file) | set char mood (file));
      0
    else up bfileprotect; undefined
    fi
  end ;
```

c)
```
proc create = (ref file file, channel chan) int :
  begin pos max pos = max pos of chan;
    establish (file, skip, chan, p of max pos, l of max pos,
      c of max pos)
  end ;
```

d)
```
proc open = (ref file file, string idf, channel chan) int :
  begin
    down bfileprotect;
    if file available (chan)
    then ref ref bfile bf := chainbfile; bool found := false;
      while (ref bfile (bf) :≠: nil) ∧ ¬ found
```

```
        do
            if match (idf, chan, book of bf)
            then found := true
            else bf := next of bf
            fi
        od;
        if ¬ found
        then up bfileprotect; undefined
        else ref book book := book of bf;
            if putting of book ∨ (put of chan) (book) ∧ users of book > 0
            then
                up bfileprotect; undefined ¢ in this case opening is
                    inhibited by other users - the system may either
                    wait, or yield nonzero (indicating unsuccessful
                    opening) immediately ¢
            else
                users of book +:= 1;
                ((put of chan) (book) | putting of book := true);
                ref ref bfile (bf) := next of bf; ¢ remove bfile from chain ¢
                (opened of file | up gremlins | up bfileprotect);
                file :=
                    (book, text of book, chan, skip, skip,
                    ¢ state: ¢ heap bool := false, heap bool := false,
                        heap bool := false, heap bool := false,
                        heap bool := true,
                    heap pos := (1, 1, 1), "", (standconv of chan) (book),
                    ¢ event routines: ¢ false, false, false, false, false,
                        false, (ref file f, ref char a) bool : false);
                (¬ bin possible (file) | set char mood (file));
                (¬ get possible (file) | set write mood (file));
                (¬ put possible (file) | set read mood (file));
                0
            fi
        fi
    else up bfileprotect; undefined
    fi
end;
```

e)

```
proc associate =
    (ref file file, ref [ ] [ ] [ ] char sss) void :
    if int p = lwb sss; int l = lwb sss [p]; int c = lwb sss [p] [l];
        p = 1 ∧ l = 1 ∧ c = 1
    then
        proc t = (ref book a) bool : true;
        proc f = (ref book a) bool : false;
        channel chan = (t, t, t, t, f, f, f, bool : false,
            pos : (max int, max int, max int), skip, skip);
        (opened of file | down bfileprotect; up gremlins);
```

```
        file :=
            (heap book := (skip, (upb sss + 1, 1, 1), skip, true, 1), sss, chan,
            skip, skip,
            ¢ state: ¢ heap bool := false, heap bool := false,
                heap bool := true, heap bool := false, heap bool := true,
            heap pos := (1, 1, 1), "", skip,
            ¢ event routines: ¢ false, false, false, false, false, false,
                (ref file f, ref char a) bool : false)
    else undefined
    fi ;
```

f)
```
    proc ? file available = (channel chan) bool :
        c true if another file, at this instant of time, may be opened on
            'chan' and false otherwise c ;
```

g)
```
    proc ? idf ok = (string idf) bool :
        c true if 'idf' is acceptable to the implementation as the
            identification of a new book and false otherwise c ;
```

h)
```
    proc ? match =
        (string idf, channel chan, ref book book name) bool :
        c true if the book referred to by 'book name' may be identified by
            'idf', and if the book may legitimately be accessed through
            'chan', and false otherwise c ;
```

i)
```
    proc ? false = (ref file file) bool : false
        ¢ this is included for brevity in 'establish', 'open' and 'associate' ¢ ;
```

j)
```
    proc ? set write mood = (ref file f) void :
        if ¬ put possible (f) ∨
            ¬ set possible (f) ∧ bin mood of f ∧ read mood of f
        then undefined
        else ref bool (read mood of f) := false; ref bool (write mood of f) := true
        fi ;
```

k)
```
    proc ? set read mood = (ref file f) void :
        if ¬ get possible (f) ∨
            ¬ set possible (f) ∧ bin mood of f ∧ write mood of f
        then undefined
        else ref bool (read mood of f) := true; ref bool (write mood of f) := false
        fi ;
```

l)
```
    proc ? set char mood = (ref file f) void :
        if ¬ set possible (f) ∧ bin mood of f
        then undefined
        else ref bool (char mood of f) := true; ref bool (bin mood of f) := false
        fi ;
```

```
m)  proc ? set bin mood = (ref file f) void :
      if ¬ bin possible (f) ∨ ¬ set possible (f) ∧ char mood of f
      then undefined
      else ref bool (char mood of f) := false; ref bool (bin mood of f) := true
      fi ;

n)  proc close = (ref file file) void :
      if opened of file
      then
          down bfileprotect;
              ref bool (opened of file) := false;
              ref book book = book of file;
              putting of book := false; users of book −:= 1;
              (text of file | (flextext): chainbfile :=
                  PRIM bfile := (book, chainbfile));
          up gremlins
      fi ;

o)  proc lock = (ref file file) void :
      if opened of file
      then
          down bfileprotect;
              ref bool (opened of file) := false;
              ref book book = book of file;
              putting of book := false; users of book −:= 1;
              (text of file | (flextext): lockedbfile :=
                  PRIM bfile := (book, lockedbfile));
          up gremlins
      fi ;

p)  proc scratch = (ref file file) void :
      if opened of file
      then
          down bfileprotect;
              ref bool (opened of file) := false;
              putting of book of file := false;
              users of book of file −:= 1;
          up gremlins
      fi ;
```

10.3.1.5. Position enquiries

(aa) The "current position" of a book opened on a given file is the value referred to by the *cpos* field of that file. It is advanced by each transput operation in accordance with the number of characters written or read.

If c is the current character number and lb is the length of the current line, then at all times $1 \leq c \leq lb + 1$. $c = 1$ implies that the next

transput operation will be to the first character of the line and c = lb + 1 implies that the line has overflowed and that the next transput operation will call an event routine. If lb = 0, then the line is empty and is therefore always in the overflowed state. Corresponding restrictions apply to the current line and page numbers. Note that, if the page has overflowed, the current line is empty and, if the book has overflowed, the current page and line are both empty (e).

bb) The user may determine the current position by means of the routines *char number*, *line number* and *page number* (a, b, c).

cc) If the current position has overflowed the line, page or book, then it is said to be outside the "physical file" (f, g, h).

dd) If, on reading, the current position is at the logical end, then it is said to be outside the "logical file" (i).}

{Each routine in this section calls *undefined* if the file is not open on entry.}

a)
```
proc char number = (ref file f) int :
    (opened of f | c of cpos of f | undefined) ;
```

b)
```
proc line number = (ref file f) int :
    (opened of f | l of cpos of f | undefined) ;
```

c)
```
proc page number = (ref file f) int :
    (opened of f | p of cpos of f | undefined) ;
```

d)
```
proc ? current pos = (ref file f) pos :
    (opened of f | cpos of f | undefined ; skip) ;
```

e)
```
proc ? book bounds = (ref file f) pos :
    begin pos cpos = current pos (f);
        int p = p of cpos, l = l of cpos;
        case text of f in
            (text t1):
                (int pb = upb t1;
                int lb = (p ≤ 0 ∨ p > pb | 0 | upb t1 [p]);
                int cb = (l ≤ 0 ∨ l > lb | 0 | upb t1 [p] [l]);
                (pb, lb, cb)),
            (flextext t2):
                (int pb = upb t2;
                int lb = (p ≤ 0 ∨ p > pb | 0 | upb t2 [p]);
                int cb = (l ≤ 0 ∨ l > lb | 0 | upb t2 [p] [l]);
                (pb, lb, cb))
        esac
    end ;
```

f)
```
proc ? line ended = (ref file f) bool :
    (int c = c of current pos (f); c > c of book bounds (f)) ;
```

g) ***proc*** ? *page ended* = (***ref file*** *f*) ***bool*** :
(***int*** *l* = *l* ***of*** *current pos (f)*; *l* > *l* ***of*** *book bounds (f))* ;

h) ***proc*** ? *physical file ended* = (***ref file*** *f*) ***bool*** :
(***int*** *p* = *p* ***of*** *current pos (f)*; *p* > *p* ***of*** *book bounds (f))* ;

i) ***proc*** ? *logical file ended* = (***ref file*** *f*) ***bool*** :
¬ (*lpos* ***of*** *book* ***of*** *f* ***beyond*** *current pos (f))* ;

10.3.1.6. Layout routines

(aa) A book input from an external medium by some **system-task** may contain lines and pages not all of the same length. Contrariwise, the lines and pages of a book which has been established (10.3.1.4.cc) are all initially of the size specified by the user. However if, during output to a compressible book (10.3.1.3.ff), *newline* (*newpage*) is called with the current position in the same line (page) as the logical end of the book, then that line (the page containing that line) is shortened to the character number (line number) of the logical end. Thus *print ((*"abcde", *newline))* could cause the current line to be reduced to 5 characters in length. Note that it is perfectly meaningful for a line to contain no characters and for a page to contain no lines.

Although the effect of a channel whose books are both compressible and of random access (10.3.1.3.ff) is well defined, it is not anticipated that such a combination is likely to occur in actual implementations.

bb) The routines *space* (a), *newline* (c) and *newpage* (d) serve to advance the current position to the next character, line or page, respectively. They do not, however, (except as provided in cc below) alter the contents of the positions skipped over. Thus *print ((*"a", *backspace, space))* has a different effect from *print ((*"a", *backspace, blank))*.

The current position may be altered also by calls of *backspace* (b), *set char number* (k) and, on appropriate channels, of *set* (i) and *reset* (j).

cc) The contents of a newly established book are undefined and both its current position and its logical end are at (1, 1, 1). As output proceeds, it is filled with characters and the logical end is moved forward accordingly. If, during character output with the current position at the logical end of the book, *space* is called, then a space character is written (similar action being taken in the case of *newline* and *newpage* if the book is not compressible).

A call of *set* which attempts to leave the current position beyond the logical end results in a call of *undefined* (a sensible system action might then be to advance the logical end to the current position, or even to the physical end of the book). There is thus no defined way in which the current position can be made to be beyond the logical end, nor in which any character within the logical file can remain in its initial undefined state.

dd) A reading or writing operation, or a **call** of *space*, *newline*, *newpage*, *set* or *set char number*, may bring the current position outside the physical or logical file (10.3.1.5.cc, dd), but this does not have any immediate consequence. However, before any further transput is attempted, or a further call of *space*, *newline* or *newpage* (but not of *set* or *set char number*) is made, the current position must be brought to a "good" position. The file is "good" if, on writing (reading), the current position is not outside the physical (logical) file (10.3.1.5.cc, dd). The page (line) is "good" if the line number (character number) has not overflowed. The event routine (10.3.1.3.cc) corresponding to *on logical file end*, *on physical file end*, *on page end* or *on line end* is therefore called as appropriate. Except in the case of formatted transput (which uses *check pos*, 10.3.3.2.c), the default action, if the event routine returns false, is to call, respectively, *undefined*, *undefined*, *newpage* or *newline*. After this (or if true is returned), if the position is still not good, an event routine (not necessarily the same one) is called again.

ee) The state of the file (10.3.1.3.bb) controls some effects of the layout routines. If the read/write mood is reading, the effect of *space*, *newline* and *newpage*, upon attempting to pass the logical end, is to call the event routine corresponding to *on logical file end* with default action *undefined*; if it is writing, the effect is to output spaces (or, in bin mood, to write some undefined character) or to compress the current line or page (see cc). If the read/write mood is not determined on entry to a layout routine, *undefined* is called. On exit, the read/write mood present on entry is restored.}

a)
```
proc space = (ref file f) void :
    if ¬ opened of f then undefined
    else
        bool reading =
            (read mood of f | true |: write mood of f | false
            | undefined; skip);
        (¬ get good line (f, reading) | undefined);
        ref pos cpos = cpos of f;
        if reading then c of cpos +:= 1
        else
            if logical file ended (f) then
                if bin mood of f then
                    (text of f | (flextext t2):
                        t2 [p of cpos ] [ l of cpos ] [ c of cpos ] := skip);
                    c of cpos +:= 1; lpos of book of f := cpos
                else put char (f, "␣")
                fi
            else c of cpos +:= 1
            fi
        fi
    fi ;
```

b)
```
proc backspace = (ref file f) void :
  if ¬ opened of f then undefined
  else ref int c = c of cpos of f;
    (c > 1 | c −:= 1 | undefined)
  fi ;
```

c)
```
proc newline = (ref file f) void :
  if ¬ opened of f then undefined
  else
    bool reading =
      (read mood of f | true |: write mood of f | false
      | undefined; skip);
    (¬ get good page (f, reading) | undefined);
    ref pos cpos = cpos of f, lpos = lpos of book of f;
    if p of cpos = p of lpos ∧ l of cpos = l of lpos
    then c of cpos := c of lpos;
      if reading then newline (f)
      else
        if compressible (f)
        then ref int pl = p of lpos, ll = l of lpos;
          flextext text = (text of f | (flextext t2): t2);
          text [pl] [ll] := text [pl] [ll] [ : c of lpos − 1 ]
        else while ¬ line ended (f) do space (f) od
        fi;
        cpos := lpos := (p of cpos, l of cpos + 1, 1)
      fi
    else cpos := (p of cpos, l of cpos + 1, 1)
    fi
  fi ;
```

d)
```
proc newpage = (ref file f) void :
  if ¬ opened of f then undefined
  else
    bool reading =
      (read mood of f | true |: write mood of f | false | undefined;
      skip);
    (¬ get good file (f, reading) | undefined);
    ref pos cpos = cpos of f, lpos = lpos of book of f;
    if p of cpos = p of lpos
    then cpos := lpos;
      if reading then newpage (f)
      else
        if compressible (f) ∧ l of lpos ≤ l of book bounds (f)
        then ref int pl = p of lpos, ll = l of lpos;
          flextext text = (text of f | (flextext t2): t2);
          text [pl] [ll] := text [pl] [ll] [ : c of lpos − 1 ];
          text [pl] := text [pl] [ : (c of lpos > 1 | ll | ll − 1) ]
```

```
            else while ¬ page ended (f) do newline (f) od
            fi;
            cpos := lpos := (p of cpos + 1, 1, 1)
        fi
    else cpos := (p of cpos + 1, 1, 1)
    fi
fi;
```

{Each of the following 3 routines either returns true, in which case the line, page or file is good (dd), or it returns false, in which case the current position may be outside the logical file or the page number may have overflowed, or it loops until the matter is resolved, or it is terminated by a **jump**. On exit, the read/write mood is as determined by its *reading* parameter.}

e)
```
proc ? get good line = (ref file f, bool reading) bool :
  begin bool not ended;
    while not ended := get good page (f, reading);
      line ended (f) ∧ not ended
    do (¬ (line mended of f) (f) | set mood (f, reading); newline (f)) od;
    not ended
  end;
```

f)
```
proc ? get good page = (ref file f, bool reading) bool :
  begin bool not ended;
    while not ended := get good file (f, reading);
      page ended (f) ∧ not ended
    do (¬ (page mended of f) (f) | set mood (f, reading); newpage (f)) od;
    not ended
  end;
```

g)
```
proc ? get good file = (ref file f, bool reading) bool :
  begin bool not ended := true;
    while set mood (f, reading);
      not ended ∧
      (reading | logical file ended | physical file ended) (f)
    do not ended := (reading | logical file mended of f
      | physical file mended of f) (f)
    od;
    not ended
  end;
```

h)
```
proc ? set mood = (ref file f, bool reading) void :
  (reading | set read mood (f) | set write mood (f));
```

i)
```
proc set = (ref file f, int p, l, c) void :
  if ¬ opened of f ∨ ¬ set possible (f) then undefined
  else bool reading =
    (read mood of f | true |: write mood of f | false | undefined; skip);
```

```
        ref pos cpos = cpos of f, lpos = lpos of book of f;
        pos ccpos = cpos;
        if (cpos := (p, l, c)) beyond lpos
        then cpos := lpos;
            (¬ (logical file mended of f) (f) | undefined);
            set mood (f, reading)
        elif pos bounds = book bounds (f);
            p < 1 ∨ p > p of bounds + 1
            ∨ l < 1 ∨ l > l of bounds + 1
            ∨ c < 1 ∨ c > c of bounds + 1
        then cpos := ccpos; undefined
        fi
    fi;
```

j)
```
    proc reset = (ref file f) void :
        if ¬ opened of f ∨ ¬ reset possible (f) then undefined
        else
            ref bool (read mood of f) := ¬ put possible (f);
            ref bool (write mood of f) := ¬ get possible (f);
            ref bool (char mood of f) := ¬ bin possible (f);
            ref bool (bin mood of f) := false;
            ref pos (cpos of f) := (1, 1, 1)
        fi;
```

k)
```
    proc set char number = (ref file f, int c) void :
        if ¬ opened of f then undefined
        else ref ref pos cpos = cpos of f;
            while c of cpos ≠ c
            do
                if c < 1 ∨ c > c of book bounds (f) + 1
                then undefined
                elif c > c of cpos
                then space (f)
                else backspace (f)
                fi
            od
        fi;
```

10.3.2. Transput values

10.3.2.1. Conversion routines

{The routines *whole*, *fixed* and *float* are intended to be used with the formatless output routines *put*, *print* and *write* when it is required to have a little extra control over the layout produced. Each of these routines has a *width* parameter whose absolute value specifies the length of the string to be produced by conversion of the arithmetic value V provided. Each of *fixed* and *float* has an *after* parameter to specify the number of digits required after the decimal point, and an *exp* parameter in *float* specifies the width allowed for the exponent. If V cannot be expressed as a string

within the given *width*, even when the value of *after*, if provided, has been reduced, then a string filled with *errorchar* (10.2.1.t) is returned instead.

Leading zeroes are replaced by spaces and a sign is normally included. The user can, however, specify that a sign is to be included only for negative values by specifying a negative width. If the width specified is zero, then the shortest possible string into which V can be converted, consistently with the other parameters, is returned. The following examples illustrate some of the possibilities:

*print (whole (i, −4))*
which might print *"␣␣␣0"*, *"␣␣99"*, *"␣−99"*, *"9999"* or, if *i* were greater than 9999, *"****"*, where *"*"* is the yield of *errorchar*;

*print (whole (i, 4))*
which would print *"␣+99"* rather than *"␣␣99"*;

*print (whole (i, 0))*
which might print *"0"*, *"99"*, *"−99"*, *"9999"* or *"99999"*;

*print (fixed (x, −6, 3))*
which might print *"␣2.718"*, *"27.183"* or *"271.83"* (in which one place after the decimal point has been sacrificed in order to fit the number in);

*print (fixed (x, 0, 3))*
which might print *"2.718"*, *"27.183"* or *"271.828"*;

*print (float (x, 9, 3, 2))*
which might print *"−2.718$_{10}$+0"*, *"+2.718$_{10}$−1"*, or *"+2.72$_{10}$+11"* (in which one place after the decimal point has been sacrificed in order to make room for the unexpectedly large exponent).]

a)
```
mode ? number = union (⦃L real⦄, ⦃L int⦄);
```

b)
```
proc whole = (number v, int width) string :
    case v in
        ⦃(L int x):
            (int length := abs width − (x < L 0 ∨ width > 0 | 1 | 0),
            L int n := abs x;
            if width = 0 then
                L int m := n; length := 0;
                while m ÷:= L 10; length +:= 1; m ≠ L 0
                do skip od
            fi;
            string s := subwhole (n, length);
            if length = 0 ∨ char in string (errorchar, loc int, s)
            then abs width × errorchar
            else
                (x < L 0 | "−" |: width > 0 | "+" | "") plusto s;
                (width ≠ 0 | (abs width − upb s) × "␣" plusto s);
                s
            fi)⦄,
        ⦃(L real x): fixed (x, width, 0)⦄
    esac ;
```

c)
```
proc fixed = (number υ, int width, after) string :
  case υ in
    ↰(L real x):
      if int length := abs width − (x < L 0 ∨ width > 0 | 1 | 0);
        after ≥ 0 ∧ (length > after ∨ width = 0)
      then L real y = abs x;
        if width = 0
        then length := (after = 0 | 1 | 0 );
          while y + L .5 × L .1 ↑ after ≥ L 10 ↑ length
          do length +:= 1 od;
          length +:= (after = 0 | 0 | after + 1)
        fi;
        string s := subfixed (y, length, after);
        if ¬ char in string (errorchar, loc int, s)
        then (length > upb s ∧ y < L 1.0 | "0" plusto s);
          (x < L 0 | "−" |: width > 0 | "+" | "") plusto s;
          (width ≠ 0 | (abs width − upb s) × "." plusto s);
          s
        elif after > 0
        then fixed (υ, width, after − 1)
        else abs width × errorchar
        fi
      else undefined; abs width × errorchar
      fi↱ ,
    ↰(L int x): fixed (L real (x), width, after)↱
  esac ;
```

d)
```
proc float = (number υ, int width, after, exp) string :
  case υ in
    ↰(L real x):
      if int before = abs width − abs exp − (after ≠ 0 | after + 1 | 0) − 2;
        sign before + sign after > 0
      then string s, L real y := abs x, int p := 0;
        L standardize (y, before, after, p);
        s :=
          fixed (sign x × y, sign width × (abs width − abs exp − 1),
            after) + "10" + whole (p, exp);
        if exp = 0 ∨ char in string (errorchar, loc int, s)
        then
          float (x, width, (after ≠ 0 | after − 1 | 0),
            (exp > 0 | exp + 1 | exp − 1))
        else s
        fi
      else undefined; abs width × errorchar
      fi↱ ,
    ↰(L int x): float (L real (x), width, after, exp)↱
  esac ;
```

e)
```
proc ? subwhole = (number υ, int width) string :
        ¢ returns a string of maximum length 'width' containing a
            decimal representation of the positive integer 'υ' ¢
    case υ in
        ⫲(L int x):
            begin string s, L int n := x;
                while dig char (S (n mod L 10)) plusto s;
                    n ÷:= L 10; n ≠ L 0
                do skip od;
                (upb s > width | width × errorchar | s)
            end ⫳
    esac ;
```

f)
```
proc ? subfixed = (number υ, int width, after) string :
        ¢ returns a string of maximum length 'width' containing a
            rounded decimal representation of the positive real number
            'υ'; if 'after' is greater than zero, this string contains a
            decimal point followed by 'after' digits ¢
    case υ in
        ⫲(L real x):
            begin string s, int before := 0;
                L real y := x + L .5 × L .1 ↑ after;
                proc choosedig = (ref L real y) char :
                    dig char ((int c := S entier (y ×:= L 10.0); (c > 9 | c := 9);
                        y −:= K c; c));
                while y ≥ L 10.0 ↑ before do before +:= 1 od;
                y /:= L 10.0 ↑ before;
                to before do s plusab choosedig (y) od;
                (after > 0 | s plusab ".");
                to after do s plusab choosedig (y) od;
                (upb s > width | width × errorchar | s)
            end ⫳
    esac ;
```

g)
```
proc ? L standardize = (ref L real y, int before, after, ref int p) void :
        ¢ adjusts the value of 'y' so that it may be transput according
            to the format $ n(before)d . n(after)d $; 'p' is set so that
            y × 10 ↑ p is equal to the original value of 'y' ¢
    begin
        L real g = L 10.0 ↑ before; L real h = g × L .1;
        while y ≥ g do y ×:= L .1; p +:= 1 od;
        (y ≠ L 0.0 | while y < h do y ×:= L 10.0; p −:= 1 od);
        (y + L .5 × L .1 ↑ after ≥ g | y := h; p +:= 1)
    end ;
```

h)
```
proc ? dig char = (int x) char : "0123456789abcdef" | x + 1 | ;
```

```
i)  proc ? string to L int = (string s, int radix, ref L int i) bool :
        ¢ returns true if the absolute value of the result is ≤ L max int ¢
    begin
        L int lr = K radix; bool safe := true;
        L int n := L 0, L int m = L max int ÷ lr;
        L int m1 = L max int - m × lr;
        for i from 2 to upb s
        while L int dig = K char dig (s [ i ]);
            safe := n < m ∨ n = m ∧ dig ≤ m1
        do n := n × lr + dig od;
        if safe then i := (s [ 1 ] = "+" | n | - n); true else false fi
    end ;
j)  proc ? string to L real = (string s, ref L real r) bool :
        ¢ returns true if the absolute value of the result is ≤
            L max real ¢
    begin
        int e := upb s + 1;
        char in string ("10", e, s);
        int p := e; char in string (".", p, s);
        int j := 1, length := 0, L real x := L 0.0;
        ¢ skip leading zeroes: ¢
        for i from 2 to e - 1
        while s [ i ] = "0" ∨ s [ i ] = "." ∨ s [ i ] = "_"
        do j := i od;
        for i from j + 1 to e - 1 while length < L real width
        do
            if s [ i ] ≠ "."
            then x := x × L 10.0 + K char dig (s [ j := i ]); length +:= 1
            fi ¢ all significant digits converted ¢
        od;
        ¢ set preliminary exponent: ¢
        int exp := (p > j | p - j - 1 | p - j), expart := 0;
        ¢ convert exponent part: ¢
        bool safe :=
            if e < upb s
            then string to L int (s [ e + 1 : ], 10, expart)
            else true
            fi;
        ¢ prepare a representation of L max real to compare with the
            L real value to be delivered: ¢
        L real max stag := L max real, int max exp := 0;
        L standardize (max stag, length, 0, max exp); exp +:= expart;
        if ¬ safe ∨ (exp > max exp ∨ exp = max exp ∧ x > max stag)
        then false
        else r := (s [ 1 ] = "+" | x | - x) × L 10.0 ↑ exp; true
        fi
    end ;
```

k) **proc** ? *char dig* = (**char** *x*) **int** :
*(x = "₋" | 0 | **int** i; char in string (x, i, "0123456789abcdef"); i − 1) ;*

l) **proc** *char in string* = (**char** *c*, **ref int** *i*, **string** *s*) **bool** :
*(**bool** found := **false**;*
*for k from lwb s to upb s while ¬ found*
*do (c = s [k] | i := k; found := **true**) od;*
*found) ;*

m) **int** *L int width* =
¢ *the smallest integral value such that 'L max int' may be converted without error using the pattern n(L int width)d* ¢
*(**int** c := 1;*
***while L** 10 ↑ (c − 1) < **L** .1 × L max int **do** c +:= 1 **od**;*
*c) ;*

n) **int** *L real width* =
¢ *the smallest integral value such that different strings are produced by conversion of '1.0' and of '1.0 + L small real' using the pattern d . n(L real width − 1)d* ¢
*1 − **S entier** (L ln (L small real) / L ln (**L** 10)) ;*

o) **int** *L exp width* =
¢ *the smallest integral value such that 'L max real' may be converted without error using the pattern d . n(L real width − 1)d e n(L exp width)d* ¢
*1 + **S entier** (L ln (L ln (L max real) / L ln (**L** 10)) / L ln (**L** 10)) ;*

10.3.2.2. Transput modes

a) ***mode*** ? ***simplout*** = ***union*** (⦃***L int***⦄, ⦃***L real***⦄, ⦃***L compl***⦄, ***bool***, ⦃***L bits***⦄, ***char***, [ ] ***char***) ;

b) ***mode*** ? ***outtype*** = ***c*** *an actual-declarer specifying a mode united from* {2.1.3.6.a} *a sufficient set of modes none of which is* **'void'** *or contains* **'flexible'**, **'reference to'**, **'procedure'** *or* **'union of' c** ;

c) ***mode*** ? ***simplin*** = ***union*** (⦃***ref L int***⦄, ⦃***ref L real***⦄, ⦃***ref L compl***⦄, ***ref bool***, ⦃***ref L bits***⦄, ***ref char***, ***ref*** [ ] ***char***, ***ref string***) ;

d) ***mode*** ? ***intype*** = ***c*** *an actual-declarer specifying a mode united from* {2.1.3.6.a} **'reference to flexible row of character'** *together with a sufficient set of modes each of which is* **'reference to'** *followed by a mode which does not contain* **'flexible'**, **'reference to'**, **'procedure'** *or* **'union of' c** ;

{See the remarks after 10.2.3.1 concerning the term "sufficient set".}

10.3.2.3. Straightening

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

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

c) The result of "straightening" a given value V is a multiple value W {of one dimension} obtained as follows:
- it is required that V {if it is a name} be not nil;
- a counter i is set to 0;
- V is "traversed" {d} using i;
- W is composed of a descriptor ((1, i)) and the elements obtained by traversing V;
- if V is not (is) a name, then the mode of the result is the mode specified by [ ] ***simplout*** ([ ] ***simplin***).

d) A value V is "traversed", using a counter i, as follows:
If V is (refers to) a value from whose mode that specified by ***simplout*** is united,
then
- i is increased by one;
- the element of W selected by (i) is V;

otherwise,
Case A: V is (refers to) a multiple value {of one dimension} having a descriptor ((l, u)):
- for j = l, ... , u, the element (the subname) of V selected by (j) is traversed using i;

Case B: V is (refers to) a multiple value {of n dimensions, n ≥ 2} whose descriptor is $((l_1, u_1), (l_2, u_2), \ldots, (l_n, u_n))$ where n ≥ 2:
- for j = $l_1$, ... , $u_1$, the multiple value selected {2.1.3.4.i} by (the name generated {2.1.3.4.j} by) the trim $(j, (l_2, u_2, 0), \ldots, (l_n, u_n, 0))$ is traversed using i;

Case C: V is (refers to) a structured value V1:
- the fields (the subnames of V referring to the fields) of V1, taken in order, are traversed using i.

### 10.3.3. Formatless transput

{In formatless transput, the elements of a "data list" are transput, one after the other, via a specified file. Each element of the data list is either a layout routine of the mode specified by ***proc*** ***(ref file) void*** (10.3.1.6) or a value of the mode specified by ***outtype*** (on output) or ***intype*** (on input). On encountering a layout routine in the data list, that routine is called with the specified file as parameter. Other values in the data list are first straightened (10.3.2.3) and the resulting values are then transput via the given file one after the other.

Transput normally takes place at the current position but, if there is no room on the current line (on output) or if a readable value is not present there (on input), then first, the event routine corresponding to *on line end*

(or, where appropriate, to *on page end*, *on physical file end* or *on logical file end*) is called, and next, if this returns false, the next "good" character position of the book is found, viz., the first character position of the next nonempty line.}

10.3.3.1. Formatless output

{For formatless output, *put* (a) and *print* (or *write*) (10.5.1.d) may be used. Each straightened value V from the data list is output as follows:

aa) If the mode of V is specified by ***L int***, then first, if there is not enough room for *L int width + 2* characters on the remainder of the current line, a good position is found on a subsequent line (see 10.3.3); next, when not at the beginning of a line, a space is given and then V is output as if under the control of the **picture** *n(L int width − 1)z + d*.

bb) If the mode of V is specified by ***L real***, then first, if there is not enough room for ***L*** *real width + L exp width + 5* characters on the current line, then a good position is found on a subsequent line; next, when not at the beginning of a line, a space is given and then V is output as if under control of the **picture**

*+d . n(L real width − 1)den(L exp width − 1)z +d*.

cc) If the mode of V is specified by ***L compl***, then first, if there is not enough room for *2 × (L real width + L exp width) + 11* characters on the current line, then a good position is found on a subsequent line; next, when not at the beginning of a line, a space is given and then V is output as if under control of the **picture**

*+d . n(L real width − 1)den(L exp width − 1)z +d"i"*
*+d . n(L real width − 1)den(L exp width − 1)z +d*.

dd) If the mode of V is specified by ***bool***, then first, if the current line is full, a good position is found on a subsequent line; next, if V is true (false), the character yielded by *flip* (*flop*) is output (with no intervening space).

ee) If the mode of V is specified by ***L bits***, then the elements of the only field of V are output (as in dd) one after the other (with no intervening spaces, and with new lines being taken as required).

ff) If the mode of V is specified by ***char***, then first, if the current line is full, a good position is found on a subsequent line; next V is output (with no intervening space).

gg) If the mode of V is specified by [ ] ***char***, then the elements of V are output (as in ff) one after the other (with no intervening spaces, and with new lines being taken as required).}

a)

```
proc put = (ref file f, [ ] union (outtype, proc (ref file) void) x) void :
  if opened of f then
    for i to upb x
    do case set write mood (f); set char mood (f); x [i] in
      (proc (ref file) void pf): pf (f),
      (outtype ot):
      begin
        [ ] simplout y = straightout ot;
        ¢proc L real conv = (L real r) string :
          float (r, L real width + L exp width + 4,
              L real width - 1, L exp width + 1)¢;
        for j to upb y
        do case y [j] in
          (union (number, ¢L compl¢) nc):
            begin string s :=
              case nc in
                ¢(L int k): whole (k, L int width + 1)¢,
                ¢(L real r): L real conv (r)¢,
                ¢(L compl z): L real conv (re z) + "±⊥"
                  + L real conv (im z)¢
              esac;
              ref ref pos cpos = cpos of f, int n = upb s;
              while
                next pos (f);
                (n > c of book bounds (f) | undefined);
                c of cpos + (c of cpos = 1 | n | n + 1) >
                c of book bounds (f) + 1
              do (¬ (line mended of f) (f) | put (f, newline));
                set write mood (f)
              od;
              (c of cpos ≠ 1 | "±" plusto s);
              for k to upb s do put char (f, s [k]) od
            end ¢ numeric ¢,
          (bool b): (next pos (f); put char (f, (b | flip | flop))),
          ¢(L bits lb):
            for k to L bits width
            do put (f, (L F of lb) [k]) od¢,
          (char k): (next pos (f); put char (f, k)),
          ([ ] char ss):
            for k from lwb ss to upb ss
            do next pos (f); put char (f, ss [k]) od
        esac od
      end
    esac od
  else undefined
  fi;
```

b)
```
proc ? put char = (ref file f, char char) void :
  if opened of f ∧ ¬ line ended (f)
  then ref pos cpos = cpos of f, lpos = lpos of book of f;
    set char mood (f); set write mood (f);
    ref int p = p of cpos, l = l of cpos, c = c of cpos;
    char k; bool found := false;
    case text of f in
      (text): (k := char; found := true),
      (flextext):
        for i to upb F of conv of f while ¬ found
        do struct (char internal, external) key = (F of conv of f) [i];
          (internal of key = char | k := external of key;
            found := true)
        od
    esac;
    if found then
      case text of f in
        (text t1): t1 [p] [l] [c] := k,
        (flextext t2): t2 [p] [l] [c] := k
      esac;
      c +:= 1;
      if cpos beyond lpos then lpos := cpos
      elif ¬ set possible (f) ∧ pos (p of lpos, l of lpos, 1) beyond cpos
      then lpos := cpos;
        (compressible (f) |
          c the size of the line and page containing the logical
            end of the book and of all subsequent lines and
            pages may be increased {e.g., to the sizes with
            which the book was originally established
            (10.3.1.4.cc) or to the sizes implied by max pos of
            chan of f} c)
      fi
    else k := "±";
      if ¬ (char error mended of f) (f, k)
      then undefined; k := "±"
      fi;
      check pos (f); put char (f, k)
    fi
  else undefined
  fi ¢ write mood is still set ¢ ;
```

c)
```
proc ? next pos = (ref file f) void :
  (¬ get good line (f, read mood of f) | undefined)
    ¢ the line is now good {10.3.1.6.dd} and the read/write mood is
    as on entry ¢ ;
```

10.3.3.2. Formatless input

{For formatless input, *get* (a) and *read* (10.5.1.e) may be used. Values from the book are assigned to each straightened name N from the data list as follows:

aa) If the mode of N is specified by ***ref L int***, then first, the book is searched for the first character that is not a space (finding good positions on subsequent lines as necessary); next, the largest string is read from the book that could be "indited" (10.3.4.1.1.kk) under the control of some **picture** of the form $+n(k1)''_{\perp}''n(k2)d\,d$ or $n(k2)d\,d$ (where *k1* and *k2* yield arbitrary nonnegative integers); this string is converted to an integer and assigned to N; if the conversion is unsuccessful, the event routine corresponding to *on value error* is called.

bb) If the mode of N is specified by ***ref L real***, then first, the book is searched for the first character that is not a space (finding good positions on subsequent lines as necessary); next, the largest string is read from the book that could be indited under the control of some **picture** of the form + $n(k1)''_{\perp}''$ *n(k2)d* or *n(k2)d* followed by . *n(k3)d d* or by *ds.*, possibly followed again by *e* $n(k4)''_{\perp}''$ + $n(k5)''_{\perp}''$ *n(k6)d d* or by *e* $n(k5)''_{\perp}''$ *n(k6)d d*; this string is converted to a real number and assigned to N; if the conversion is unsuccessful, the event routine corresponding to *on value error* is called.

cc) If the mode of N is specified by ***ref L compl***, then first, a real number is input (as in bb) and assigned to the first subname of N; next, the book is searched for the first character that is not a space; next, a character is input and, if it is not "⊥" or "*i*", then the event routine corresponding to *on char error* (10.3.1.3.cc) is called, the suggestion being "⊥"; finally, a real number is input and assigned to the second subname of N.

dd) If the mode of N is specified by ***ref bool***, then first, the book is searched for the first character that is not a space (finding good positions on subsequent lines as necessary); next, a character is read; if this character is the same as that yielded by *flip* (*flop*), then true (false) is assigned to N; otherwise, the event routine corresponding to *on char error* is called, the suggestion being *flop*.

ee) If the mode of N is specified by ***ref L bits***, then input takes place (as in dd) to the subnames of N one after the other (with new lines being taken as required).

ff) If the mode of N is specified by ***ref char***, then first, if the current line is exhausted, a good position is found on a subsequent line; next, a character is read and assigned to N.

gg) If the mode of N is specified by ***ref*** [ ] ***char***, then input takes place (as in ff) to the subnames of N one after the other (with new lines being taken as required).

hh) If the mode of N is specified by ***ref string***, then characters are read until either

(i) a character is encountered which is contained in the string associated with the file by a call of the routine *make term*, or
(ii) the current line is exhausted, whereupon the event routine corresponding to *on line end* (or, where appropriate, to *on page end*, *on physical file end* or *on logical file end*) is called; if the event routine moves the current position to a good position (see 10.3.3), then input of characters is resumed.

The string consisting of the characters read is assigned to N (note that, if the current line has already been exhausted, or if the current position is at the start of an empty line or outside the logical file, then an empty string is assigned to N).]

a)
```
proc get = (ref file f, [ ] union (intype, proc (ref file) void) x) void :
    if opened of f then
    for i to upb x
    do case set read mood (f); set char mood (f); x [i] in
        (proc (ref file) void pf): pf (f),
        (intype it):
        begin
            [ ] simplin y = straightin it; char k; bool k empty;
            op ? = (string s) bool :
                    ¢ true if the next character, if any, in the current line
                    is contained in 's' (the character is assigned to 'k')
                    and false otherwise ¢
                if k empty ∧ (line ended (f) ∨ logical file ended (f))
                then false
                else (k empty | get char (f, k));
                    k empty := char in string (k, loc int, s)
                fi;
            op ? = (char c) bool : ? string (c);
            prio ! = 8;
            op ! = (string s, char c) char :
                    ¢ expects a character contained in 's'; if the character
                    read is not in 's', the event routine corresponding to
                    'on char error' is called with the suggestion 'c' ¢
                if (k empty | check pos (f); get char (f, k));
                    k empty := true;
                    char in string (k, loc int, s)
                then k
                else char sugg := c;
                    if (char error mended of f) (f, sugg) then
                        (char in string (sugg, loc int, s)
                        | sugg
                        | undefined; c)
```

```
        else undefined; c
        fi;
        set read mood (f)
    fi;
op ! = (char s, c) char : string (s) ! c;
proc skip initial spaces = void :
    while (k empty | next pos (f)); ? "␣" do skip od;
proc skip spaces = void :
    while ? "␣" do skip od;
proc read dig = string :
    (string t := "0123456789" ! "0";
    while ? "0123456789" do t plusab k od; t);
proc read sign = char :
    (char t = (skip spaces; ? "+-" | k | "+");
    skip spaces; t);
proc read num = string :
    (char t = read sign; t + read dig);
proc read real = string :
    (string t := read sign;
    (¬ ? "." | t plusab read dig | k empty := false);
    (? "." | t plusab "." + read dig);
    (? "10\e" | t plusab "10" + read num); t);


for j to upb y
do bool incomp := false; k empty := true;
    case y [j] in
    ⦃(ref L int ii):
        (skip initial spaces;
            incomp := ¬ string to L int (read num, 10, ii))⦄,
    ⦃(ref L real rr):
        (skip initial spaces;
            incomp := ¬ string to L real (read real, rr))⦄,
    ⦃(ref L compl zz):
        (skip initial spaces;
            incomp := ¬ string to L real (read real, re of zz);
            skip spaces; "i⊥" ! "⊥";
            incomp := incomp ∨
             ¬ string to L real (read real, im of zz))⦄,
    (ref bool bb):
        (skip initial spaces;
        bb := (flip + flop) ! flop = flip),
    ⦃(ref L bits lb):
        for i to L bits width
        do get (f, (L F of lb) [i]) od⦄,
    (ref char cc): (next pos (f); get char (f, cc)),
    (ref [ ] char ss):
```

```
            for i from lwb ss to upb ss
            do next pos (f); get char (f, ss [i]) od ,
        (ref string ss):
            begin string t;
            while check pos (f);
                if line ended (f) ∨ logical file ended (f)
                then false
                else get char (f, k);
                    k empty := ¬ char in string (k, loc int, term of f)
                fi
            do t plusab k od;
            ss := t
            end
        esac;
        (¬ k empty | backspace (f));
        if incomp
        then (¬ (value error mended of f) (f) | undefined);
            set read mood (f)
        fi
    od
  end
esac od
else undefined
fi ;
```

b)

```
proc ? get char = (ref file f, ref char char) void :
  if opened of f ∧ ¬ line ended (f) ∧ ¬ logical file ended (f)
  then ref pos cpos = cpos of f;
    set char mood (f); set read mood (f);
    int p = p of cpos, l = l of cpos, c = c of cpos;
    c of cpos +:= 1;
    char := case text of f in
      (text t1): t1 [p] [l] [c] ,
      (flextext t2):
        (char k := t2 [p] [l] [c];
        bool found := false;
        for i to upb F of conv of f while ¬ found
        do struct (char internal, external) key = (F of conv of f) [i];
          (external of key = k | k := internal of key; found := true)
        od;
        if found then k
        else k := "␣";
          if (char error mended of f) (f, k)
          then k
          else undefined; "␣"
          fi;
```

```
                    set read mood (f)
                fi)
            esac
        else undefined
        fi ¢ read mood is still set ¢ ;
```

c)
```
    proc ? check pos = (ref file f) void :
        begin bool reading = read mood of f;
            bool not ended := true;
            while not ended := not ended ∧ get good page (f, reading);
                line ended (f) ∧ not ended
            do not ended := (line mended of f) (f) od
        end ;
```

{The routine *check pos* is used in formatted transput before each call of *put char* or *get char*. If the position is not good (10.3.1.6.dd), it calls the appropriate event routine, and may call further event routines if true is returned. If the event routine corresponding to *on page end* returns false, *newpage* is called but, if any other event routine returns false, no default action is taken and no more event routines are called. On exit, the read/write mood is as on entry, but the current position may not be good, in which case *undefined* will be called in the following *put char* or *get char*. However, *check pos* is also called when getting strings (hh), in which case the string is then terminated if the current position is not good.}

10.3.4. Format texts

{In formatted transput, each straightened value from a data list (cf. 10.3.3) is matched against a constituent **picture** of a **format-text** provided by the user. A **picture** specifies how a value is to be converted to or from a sequence of characters and prescribes the layout of those characters in the book. Features which may be specified include the number of digits, the position of the decimal point and of the sign, if any, suppression of zeroes and the insertion of arbitrary strings. For example, using the **picture** *-d.3d "$_{\perp}$" 3d "$_{\perp}$" e z+d*, the value 1234.567 would be transput as the string *"$_{\perp}$1.234$_{\perp}$567$_{\perp 10 \perp}$+3"*.

A "format" is a structured value (i.e., an internal object) of mode **'FORMAT'**, which mirrors the hierarchical structure of a **format-text** (which is an external object). In this section are given the syntax of **format-texts** and the semantics for obtaining their corresponding formats. The actual formatted transput is performed by the routines given in section 10.3.5 but, for convenience, a description of their operation is given here, in association with the corresponding syntax.}

10.3.4.1. Collections and pictures

10.3.4.1.1. Syntax

{The following **mode-declarations** (taken from 10.3.5.a) are reflected in the metaproduction rules A to K below.

```
A)  mode format = struct (flex [1 : 0] piece F);
B)  mode piece = struct (int cp, count, bp, flex [1 : 0] collection c);
C)  mode collection =union (picture, collitem);
D)  mode collitem =
      struct (insertion i1, proc int rep, int p, insertion i2);
E)  mode insertion =
      flex [1 : 0] struct (proc int rep, union (string, char) sa);
F)  mode picture = struct
      (union (pattern, cpattern, fpattern, gpattern, void) p, insertion i);
G)  mode pattern = struct (int type, flex [1 : 0] frame frames);
H)  mode frame =
      struct (insertion i, proc int rep, bool supp, char marker);
I)  mode cpattern =

      struct (insertion i, int type, flex [1 : 0] insertion c);
J)  mode fpattern = struct (insertion i, proc format pf);
K)  mode gpattern = struct (insertion i, flex [1 : 0] proc int spec);)
```

A) FORMAT :: structured with row of PIECE field letter aleph mode.

B) PIECE :: structured with integral field letter c letter p
integral field letter c letter o letter u letter n letter t
integral field letter b letter p
row of COLLECTION field letter c mode.

C) COLLECTION :: union of PICTURE COLLITEM mode.

D) COLLITEM :: structured with INSERTION field letter i digit one
procedure yielding integral field letter r letter e letter p
integral field letter p
INSERTION field letter i digit two mode.

E) INSERTION :: row of structured with procedure yielding integral
field letter r letter e letter p
union of row of character character mode field
letter s letter a mode.

F) PICTURE :: structured with union of
PATTERN CPATTERN FPATTERN GPATTERN void mode
field letter p INSERTION field letter i mode.

G) PATTERN :: structured with
integral field letter t letter y letter p letter e
row of FRAME field
letter f letter r letter a letter m letter e letter s mode.

H) FRAME :: structured with INSERTION field letter i
procedure yielding integral field letter r letter e letter p
boolean field letter s letter u letter p letter p character field
letter m letter a letter r letter k letter e letter r mode.

I) CPATTERN :: structured with INSERTION field letter i
integral field letter t letter y letter p letter e
row of INSERTION field letter c mode.

```
J)  FPATTERN :: structured with INSERTION field letter i
        procedure yielding FIVMAT field letter p letter f mode.
K)  GPATTERN :: structured with INSERTION field letter i
        row of procedure yielding integral field
        letter s letter p letter e letter c mode.
L)  FIVMAT ::
      mui definition of structured with
        row of structured with integral field letter c letter p
          integral field letter c letter o letter u letter n letter t
          integral field letter b letter p
          row of union of
            structured with
              union of PATTERN CPATTERN
                structured with INSERTION field letter i
                  procedure yielding mui application field
                   letter p letter f
                mode
                GPATTERN void
              mode field letter p
              INSERTION field letter i
            mode
            COLLITEM
          mode field letter c
        mode field letter aleph
      mode.
    {'FIVMAT' is equivalent (2.1.1.2.a) to 'FORMAT'.}

M)  MARK :: sign ; point ; exponent ; complex ; boolean.
N)  COMARK :: zero ; digit ; character.
O)  UNSUPPRESSETY :: unsuppressible ; EMPTY.
P)  TYPE :: integral ; real ; boolean ; complex ; string ; bits ;
      integral choice ; boolean choice ; format ; general.

a)  FORMAT NEST format text{5D} : formatter{94f} token,
        NEST collection{b} list, formatter{94f} token.
b)  NEST collection{a,b} :
      pragment{92a} sequence option, NEST picture{c} ;
      pragment{92a} sequence option, NEST insertion{d},
        NEST replicator{g}, NEST collection{b} list brief pack,
        pragment{92a} sequence option, NEST insertion{d}.
c)  NEST picture{b} : NEST TYPE pattern{A342a,A343a,A344a,
          A345a,A346a,A347a,A348a,b,A349a,A34Aa} option,
        NEST insertion{d}.
d)  NEST insertion{b,c,j,k,A347b,A348a,b,A349a,A34Aa} :
      NEST literal{i} option, NEST alignment{e} sequence option.
e)  NEST alignment{d} :
      NEST replicator{g}, alignment code{f}, NEST literal{i} option.
```

f) **alignment code**{e} : **letter k**{94a} **symbol** ; **letter x**{94a} **symbol** ; **letter y**{94a} **symbol** ; **letter l**{94a} **symbol** ; **letter p**{94a} **symbol** ; **letter q**{94a} **symbol**.

g) **NEST replicator**{b,e,i,k} : **NEST unsuppressible replicator**{h} **option**.

h) **NEST unsuppressible replicator**{g,i} : **fixed point numeral**{811b} ; **letter n**{94a} **symbol**, **meek integral NEST ENCLOSED clause**{31a,34a,-}, **pragment**{92a} **sequence option**.

i) **NEST UNSUPPRESSETY literal**{d,e,i,A348c} : **NEST UNSUPPRESSETY replicator**{g,h}, **strong row of character NEST denoter**{80a} **coercee**{61a}, **NEST unsuppressible literal**{i} **option**.

j) **NEST UNSUPPRESSETY MARK frame**{A342c,A343b,c,A344a,A345a} : **NEST insertion**{d}, **UNSUPPRESSETY suppression**{l}, **MARK marker**{A342e,A343d,e,A344b,A345b}.

k) **NEST UNSUPPRESSETY COMARK frame**{A342b,c,A346a} : **NEST insertion**{d}, **NEST replicator**{g}, **UNSUPPRESSETY suppression**{l}, **COMARK marker**{A342d,f,A346b}.

l) **UNSUPPRESSETY suppression**{j,k,A347b} : **where (UNSUPPRESSETY) is (unsuppressible), EMPTY** ; **where (UNSUPPRESSETY) is (EMPTY)**, **letter s**{94a} **symbol option**.

m) ***frame** : **NEST UNSUPPRESSETY MARK frame**{j} ; **NEST UNSUPPRESSETY COMARK frame**{k} ; **NEST RADIX frame**{A347b}.

n) ***marker** : **MARK marker**{A342e,A343d,e,A344b,A345b} ; **COMARK marker**{A342d,f,A346b} ; **radix marker**{A347c}.

o) ***pattern** : **NEST TYPE pattern**{A342a,A343a,A344a,A345a, A346a,A347a,A348a,b,A349a,A34Aa}.

{Examples:

a) *$ p "table of"x 10a,l n (lim - 1) ("x=" 12z+d 2x, +.12de+2d 3q"+j×"3"±" si +.10de+2d l) p $*

b) *p "table of"x 10a • l n (lim - 1) ("x=" 12z+d 2x, +.12de+2d 3q"+j×"3"±" si +.10de+2d l) p*

c) *l 20k c ("mon", "tues", "wednes", "thurs", "fri", "satur", "sun") "day"*

d) *p "table of"x*

e) *p "table of"*

h) *10 • n (lim - 1)*

i) *"+j×"3"±"*

j) *si*

k) *"x=" 12z*

l) *s* }

{The positions where **pragments** (9.2.1.a) may occur in **format-texts** are restricted. In general (as elsewhere in the language), a **pragment** may not occur between two **DIGIT-** or **LETTER-symbols**.}

{aa) For formatted output, *putf* (10.3.5.1.a) and *printf* (or *writef*) (10.5.1.f) may be used and, for formatted input, *getf* (10.3.5.2.a) and *readf* (10.5.1.g). Each element in the data list (cf. 10.3.3) is either a format to be associated with the file or a value to be transput (thus a format may be included in the data list immediately before the values to be transput using that format).

bb) During a call of *putf* or *getf*, transput proceeds as follows:
For each element of the data list, considered in turn,
If it is a format,
then it is made to be the current format of the file by *associate format* (10.3.5.k);
otherwise, the element is straightened (10.3.2.3.c) and each element of the resulting multiple value is output (hh) or input (ii) using the next "picture" (cc, gg) from the current format.

cc) A "picture" is the yield of a **picture**. It is composed of a "pattern" of some specific **'TYPE'** (according to the syntax of the **TYPE-pattern** of that **picture**), followed by an "insertion" (ee). Patterns, apart from **'choice'**, **'format'** and **'general'** patterns, are composed of "frames", possibly "suppressed", each of which has an insertion, a "replicator" (dd), and a "marker" to indicate whether it is a *"d"*, *"z"*, *"i"* etc. frame. The frames of each pattern may be grouped into "sign moulds", "integral moulds", etc., according to the syntax of the corresponding **pattern**.

dd) A "replicator" is a routine, returning an integer, constructed from a **replicator** (10.3.4.1.2.c). For example, the **replicator** *10* gives rise to a routine composed from ***int***: *10*; moreover, *n (lim - 1)* is a "dynamic" **replicator** and gives rise to ***int***: *(lim - 1)*. Note that the scope of a replicator restricts the scope of any format containing it, and thus it may be necessary to take a local copy of a file before associating a format with it (see, e.g., 11.13). A replicator which returns a negative value is treated as if it had returned zero (*"k"* alignments apart).

When a picture is "staticized", all of its replicators and other routines (including those contained in its insertions) are called collaterally. A staticized pattern may be said to "control" a string, and there is then a correspondence between the frames of that pattern, taken in order, and the characters of the string. Each frame controls n consecutive characters of the string, where n is 0 for an *"r"* frame and, otherwise, is the integer returned by the replicator of the frame (which is always 1 for a "+", "-", ".", *"e"*, *"i"* or *"b"* frame). Each controlled character must be one of a limited set appropriate to that frame.

ee) An "insertion", which is the yield of an **insertion** (10.3.4.1.2.d), is a sequence of replicated "alignments" and strings; an insertion containing no alignments is termed a "literal". An insertion is "performed" by performing its alignments (ff) and on output (input) writing ("expecting" (ll)) each character of its replicated strings (a string is replicated by repeating it the number of times returned by its replicator).

ff) An "alignment" is the character yielded by an **alignment-code** (10.3.4.1.2.d). An alignment which has been replicated n times is performed as follows:

- *"k"* causes *set char number* to be called, with n as its second parameter;
- *"x"* causes *space* to be called n times;
- *"y"* causes *backspace* to be called n times;
- *"l"* causes *newline* to be called n times;
- *"p"* causes *newpage* to be called n times;
- *"q"* on output (input) causes the character *blank* to be written (expected) n times.

gg) A format may consist of a sequence of pictures, each of which is selected in turn by *get next picture* (10.3.5.b). In addition, a set of pictures may be grouped together to form a replicated "collection" (which may contain further such collections). When the last picture in a collection has been selected, its first picture is selected again, and so on until the whole collection has been repeated n times, where n is the integer returned by its replicator. A collection may be provided with two insertions, the first to be performed before the collection, the second afterwards.

A format may also invoke other formats by means of **'format'** patterns (10.3.4.9.1).

When a format has been exhausted, the event routine corresponding to *on format end* is called; if this returns false, the format is repeated; otherwise, if the event routine has failed to provide a new format, *undefined* is called.

hh) A value V is output, using a picture P, as follows:
If the pattern Q of P is a **'choice'** or **'general'** pattern,
then V is output using P (see 10.3.4.8.1.aa,dd, 10.3.4.10.1.aa);
otherwise, V is output as follows:

- P is staticized;

If the mode of V is "output compatible" with Q (see the separate section dealing with each type of pattern),
then

- V is converted into a string controlled (dd) by Q (see the appropriate section);

If the mode is not output compatible, or if the conversion is unsuccessful,

then
- the event routine corresponding to *on value error* is called;
- if this returns false, V is output using *put* and *undefined* is called;

otherwise, the string is "edited" (jj) using Q;
- the insertion of P is performed.

ii) A value is input to a name N, using a picture P, as follows:
If the pattern Q of P is a **'choice'** or **'general'** pattern,
then a value is input to N using P (see 10.3.4.8.1.bb,ee, 10.3.4.10.1.bb);
otherwise,
- P is staticized;
- a string controlled by Q is "indited" (kk);

If the mode of N is "input compatible" with Q (see the appropriate section),
then
- the string is converted to an appropriate value suitable for N using Q (see the appropriate section);
- if the conversion is successful, the value is assigned to N;

If the mode is not input-compatible, or if the conversion is unsuccessful,
then
- the event routine corresponding to *on value error* is called;
- if this returns false, *undefined* is called;

- the insertion of P is performed.

jj) A string is "edited", using a pattern P, as follows:
In each part of the string controlled by a sign mould,
- if the first character of the string (which indicates the sign) is *"+"* and the sign mould contains a *"−"* frame, then that character is replaced by *" "*;
- the first character (i.e., the sign) is shifted to the right across all leading zeroes in this part of the string and these zeroes are replaced by spaces (for example, using the sign mould *4z+*, the string *"+0003"* becomes *" "+3"*);

In each part of the string controlled by an integral mould,
- zeroes controlled by *"z"* frames are replaced by spaces as follows:
  - between the start of the string and the first nonzero digit;
  - between each *"d"*, *"e"* or *"i"* frame and the next nonzero digit;

(for example, using the pattern *zdzd2d*, the string *"180168"* becomes *"18 168"*;)

For each frame F of P,
- the insertion of F is performed;
- if F is not suppressed, the characters controlled by F are written;

(for example, the string *"+0003.5"*, when edited using the pattern *4z+ s. "," d*, causes the string *" "+3,5"* to be written and the string *"180168"*, using the pattern *zd"−"zd"−19"2d*, gives rise to *"18− 1−1968"*).

kk) A string is "indited", using a pattern P, as follows:
For each frame F of P,
- the insertion of F is performed;

For each element of the string controlled by F, a character is obtained as follows:
If F is contained in a sign mould,
then
- if a sign has been found, a digit is expected, with *"0"* as suggestion;
- otherwise, either a "+" or a "−" is expected, with "+" as suggestion, and, in addition, if the sign mould contains a "−" frame, then a space preceding the first digit will be accepted as the sign (and replaced by "+");

otherwise, if F is contained in an integral mould,
then
If F is suppressed,
then *"0"* is supplied;
otherwise,
Case A: F is a *"d"* frame:
- a digit is expected, with *"0"* as suggestion;

Case B: F is a *"z"* frame:
- a digit or space is expected, with *"0"* as suggestion, but a space is only acceptable as follows:
  - between the start of the string and the first nonzero digit;
  - between each *"d"*, *"e"* or *"i"* frame and the next nonzero digit;
- such spaces are replaced by zeroes;

otherwise, if F is an *"a"* frame,
then if F is not suppressed, a character is read and supplied; otherwise "␣" is supplied;
otherwise, if F is not suppressed,
then if F is a "." (*"e"*, *"i"*, *"b"*) frame, a "." ("$_{10}$" or "\" or *"e"*, "⊥" or *"i"*, *flip* or *flop*) is expected, with "." ("$_{10}$", "⊥", *flop*) as suggestion;
otherwise, if F is a suppressed "." (*"e"*, *"i"*) frame, the character "." ("$_{10}$", "⊥") is supplied.

ll) A member of a set of characters S is "expected", with the character C as suggestion, as follows:
- a character is read;

If that character is one of the expected characters (i.e., a member of S), then that character is supplied;
otherwise, the event routine corresponding to *on char error* is called, with C as suggestion; if this returns true and C, as possibly replaced, is one of the expected characters, then that character is supplied; otherwise, *undefined* is called.]

10.3.4.1.2. Semantics

{A format is brought into being by means of a **format-text**. A format is best regarded as a tree, with a collection at each node and a picture at each tip. In order to avoid violation of the scope restrictions, each node of this tree is, in this Report, packed into a value of mode **'PIECE'**. A format is composed of a row of such pieces and the pieces contain pointers to each other in the form of indices selecting from that row. An implementer will doubtless store the tree in a more efficient manner. This is possible because the **field-selector** of a format is hidden from the user in order that he may not break it open.

Although a **format-text** may contain **ENCLOSED-clauses** (in **replicators** and **format-patterns**) or **units** (in **general-patterns**), these are not elaborated at this stage but are, rather, turned into routines for subsequent calling as and when they are encountered during formatted transput. Indeed, the elaboration of a **format-text** does not result in any actions of any significance to the user.}

a) The yield of a **format-text** F, in an environ E, is a structured value whose only field is a multiple value W, whose mode is **'row of PIECE'**, composed of a descriptor ((1, n)) and n elements determined as follows:
- a counter i is set to 1;
- F is "transformed" {b} in E into W, using i.

b) A **format-text** or a **collection-list-pack** C is "transformed" in an environ E into a multiple value W whose mode is **'row of PIECE'**, using a counter i, as follows:
- the element of W selected by (i) is a structured value, whose mode is **'PIECE'** and whose fields, taken in order, are
  - {*cp*} undefined;
  - {*count*} undefined;
  - {*bp*} undefined;
  - {*c*} a multiple value V, whose mode is **'row of COLLECTION'**, having a descriptor ((1, m)), where m is the number of constituent **collections** of C, and elements determined as follows:

For j = 1, ... , m, letting $C_j$ be the j-th constituent **collection** of C,

Case A: The direct descendents of $C_j$ include a **picture** P:

- the constituent **pattern** T, if any, and the **insertion** I of P are elaborated collaterally;
- the j-th element of V is a structured value, whose mode is **'PICTURE'** and whose fields, taken in order, are
  - {*p*} the yield of T, if any, {e, 10.3.4.8.2, 10.3.4.9.2, 10.3.4.10.2} and, otherwise, empty;
  - {*i*} the yield of I {d};

Case B: The direct descendents of $C_j$ include a first **insertion** I1, a **replicator** REP, a **collection-list-pack** P and a second **insertion** I2:

- i is increased by 1;
- I1, REP and I2 are elaborated collaterally;
- the j-th element of V is a structured value whose mode is **'COLLITEM'** and whose fields, taken in order, are
  - {*i1*} the yield of I1 {d};
  - {*rep*} the yield of REP {c};
  - {*p*} i;
  - {*i2*} the yield of I2;
- P is transformed in E into W, using i.

c) The yield, in an environ E, of a **NEST-UNSUPPRESSETY-replicator** R {10.3.4.1.1.g,h} is a routine whose mode is **'procedure yielding integral'**, composed of a **procedure-yielding-integral-NEST-routine-text** whose **unit** is U, together with the environ necessary {7.2.2.c} for U in E, where U is determined as follows:

Case A: R contains a **meek-integral-ENCLOSED-clause** C:

- U is a new **unit** akin {1.1.3.2.k} to C;

Case B: R contains a **fixed-point-numeral** D, but no **ENCLOSED-clause:**

- U is a new **unit** akin to D;

Case C: R is invisible:

- U is a new **unit** akin to a **fixed-point-numeral** which has an intrinsic value {8.1.1.2} of 1.

d) The yield of an **insertion** I {10.3.4.1.1.d} is a multiple value W whose mode is **'INSERTION'**, determined as follows:

- let $U_1$, ... , $U_n$ be the constituent **UNSUPPRESSETY-replicators** of I, and let $A_i$, i = 1, ... , n, be the **denoter-coercee** or **alignment-code** {immediately} following $U_i$;
- let $R_1$, ... , $R_n$ and $D_1$, ... , $D_n$ be the {collateral} yields of $U_1$, ... , $U_n$ and $A_1$, ... , $A_n$, where the yield of an **alignment-code** is the {character which is the} intrinsic value {8.1.4.2.b} of its **LETTER-symbol**;
- the descriptor of W is ((1, n));
- the element of W selected by (i), i = 1, ... , n, is a structured value {of the mode specified by ***struct*** *(**proc int** rep, **union** (**string**, **char**) sa)*} whose fields, taken in order, are
  - {*rep*} $R_i$;
  - {*sa*} $D_i$.

e) The yield of an **integral-**, **real-**, **boolean-**, **complex-**, **string-** or **bits-pattern** P {10.3.4.2.1.a, 10.3.4.3.1.a, ... , 10.3.4.7.1.a} is a structured value W whose mode is **'PATTERN'**, determined as follows:

- let $V_1$, ... , $V_n$ be the {collateral} yields of the constituent **frames** of P {f};
- the fields of W, taken in order, are
  - {*type*} 1 (2, 3, 4, 5) if P is an **integral-** (**real-**, **boolean-**, **complex-**, **string-**) **-pattern** and 6 (8, 12, 20) if P is a **bits-pattern** whose constituent **RADIX** is a **radix-two** (**-four**, **-eight**, **-sixteen**);
  - {*frames*} a multiple value, whose mode is '**row of FRAME**', having a descriptor ((1, n)) and n elements, that selected by (i) being $V_i$.

f) The yield of a **frame** F {10.3.4.1.1.m} is a structured value W whose mode is '**FRAME**', determined as follows:
- the **insertion** and the **replicator**, if any, of F are elaborated collaterally;
- the fields of W, taken in order, are
  - {*i*} the yield of its **insertion**;
  - {*rep*} the yield of its **replicator** {c}, if any, and, otherwise, the yield of an invisible **replicator**;
  - {*supp*} true if its **UNSUPPRESSETY-suppression** contains a **letter-s-symbol** and, otherwise, false;
  - {*marker*} {the character which is} the intrinsic value {8.1.4.2.b} of a **symbol** S determined as follows:

  Case A: F is a constituent **unsuppressible-zero-frame** of a **sign-mould** {such as *3z+*} whose constituent **sign-marker** contains a **plus-symbol**:
    - S is a **letter-u-symbol**;

  Case B: F is a constituent **unsuppressible-zero-frame** of a **sign-mould** {such as *3z−*} whose constituent **sign-marker** contains a **minus-symbol**:
    - S is a **letter-v-symbol**;

  Other cases:
    - S is the constituent **symbol** of the **marker** of F.

{Thus the **zero-marker** *z* may be passed on as the character *"u"*, *"v"* or *"z"* according to whether it forms part of a **sign-mould** (with descendent **plus-symbol** or **minus-symbol**) or of an **integral-mould**.}

### 10.3.4.2. Integral patterns

#### 10.3.4.2.1. Syntax

a) **NEST integral pattern**{A341c,A343c} :
  **NEST sign mould**{c} **option**, **NEST integral mould**{b}.

b) **NEST integral mould**{a,A343b,c,A347a} :
  **NEST digit frame**{A341k} **sequence**.

c) **NEST sign mould**{a,A343a} :
  **NEST unsuppressible zero frame**{A341k} **sequence option**,
    **NEST unsuppressible sign frame**{A341j}.

d) **zero marker**{f,A341k} : **letter z**{94a} **symbol**.

e) **sign marker**{A341j} : **plus**{94c} **symbol** ; **minus**{94c} **symbol**.

f) **digit marker**{A341k} : **letter d**{94a} **symbol** ; **zero marker**{d}.

{Examples:

a) *"x="12z+d* b) *d*
c) *"x="12z+* }

{For the semantics of **integral-patterns** see 10.3.4.1.2.e.}

{aa) The modes which are output (input) compatible with an **'integral'** pattern are those specified by ***L int*** (by ***ref L int***).

bb) A value V is converted to a string S using an **'integral'** pattern P as follows:
• if P contains a sign mould, then the first character of S is the sign of V; otherwise, if V < 0, the conversion is unsuccessful;
• the remainder of S contains a decimal representation of V determined as follows:
  • the elements of S controlled by *"d"* and *"z"* frames are the appropriate digits (thus the pattern specifies the number of digits to be used);
  • if V cannot be represented by such a string, the conversion is unsuccessful.
(For example, the value 99 could be converted to a string using the pattern *zzd*, but 9999 and -99 could not.)

cc) A string S is converted to an integer suitable for a name N, using an **'integral'** pattern, as follows:
• the integer I for which S contains a decimal representation (8.1.1.2) is considered;
• if I is greater than the largest value to which N can refer, the conversion is unsuccessful; otherwise, I is the required integer (e.g., if the mode of N is specified by ***ref short int***, and the value of *short max int* is 65535, then no string containing a decimal representation of a value greater than 65535 may be converted).}

10.3.4.3. Real patterns

10.3.4.3.1. Syntax

a) **NEST real pattern**{A341c,A345a} : **NEST sign mould**{A342c} **option**,
  **NEST variable point mould**{b}
  **or alternatively NEST floating point mould**{c}.
b) **NEST variable point mould**{a,c} : **NEST integral mould**{A342b},
  **NEST point frame**{A341j}, **NEST integral mould**{A342b} **option** ;
  **NEST point frame**{A341j}, **NEST integral mould**{A342b}.
c) **NEST floating point mould**{a} :
  **NEST variable point mould**{b}
  **or alternatively NEST integral mould**{A342b},
  **NEST exponent frame**{A341j}, **NEST integral pattern**{A342a}.
d) **point marker**{A341j} : **point**{94b} **symbol**.
e) **exponent marker**{A341j} : **letter e**{94a} **symbol**.

{Examples:

a) *+zd.11d • +.12de+2d* b) *zd.11d • .12d*
c) *.12de+2d* }

{For the semantics of **real-patterns** see 10.3.4.1.2.e.}

{aa) The modes which are output (input) compatible with a **'real'** pattern are those specified by ***L real*** and ***L int*** (by ***ref L real***).

bb) A value V is converted to a string S using a **'real'** pattern P as follows:
• if P contains a sign mould, then the first character of S is the sign of V; otherwise, if V < 0, the conversion is unsuccessful;
• the remainder of S contains a decimal representation of V determined as follows:
  • if necessary, V is widened to a real number;
  • the element of S controlled by the "." ("*e*") frame, if any, of P is "." ("$_{10}$");
  If P contains an "*e*" frame,
  then
    • let W be the sequence of frames preceding, and IP be the **'integral'** pattern following, that "*e*" frame;
    • an exponent E is calculated by standardizing V to the largest value convertible using W (see below);
    • the part of S controlled by IP is obtained by converting E using IP (see 10.3.4.2.1.bb);
  otherwise,
    • let W be the whole of P;
  • the elements of S controlled by the "*d*" and "*z*" frames of W are the appropriate digits (thus the pattern specifies the number of digits to be used, and the number of digits to be placed after the decimal point, if any);
  • if V cannot be represented by such a string, the conversion is unsuccessful.

cc) A string S is converted to a real number suitable for a name N, using a **'real'** pattern, as follows:
• the real number R for which S contains a decimal representation is considered;
• if R is greater than the largest value to which N can refer, the conversion is unsuccessful; otherwise, R is the required real number.}

### 10.3.4.4. Boolean patterns

#### 10.3.4.4.1. Syntax

a) **NEST boolean pattern**{A341c} :
  **NEST unsuppressible boolean frame**{A341j}.
b) **boolean marker**{A341j,A348b} : **letter b**{94a} **symbol.**

{Example:

a) *14x b* }

{For the semantics of **boolean-patterns** see 10.3.4.1.2.e.}

{aa) The mode which is output (input) compatible with a **'boolean'** pattern is that specified by ***bool*** (***ref bool***).

bb) A value V is converted to a string using a **'boolean'** pattern as follows:
• if V is true (false), then the string is that yielded by *flip* (*flop*).

cc) A string S is converted to a boolean value, using a **'boolean'** pattern, as follows:
• if S is the same as the string yielded by *flip* (*flop*), then the required value is true (false).}

10.3.4.5. Complex patterns

10.3.4.5.1. Syntax

a) **NEST complex pattern**{A341c} **: NEST real pattern**{A343a},
**NEST complex frame**{A341j}, **NEST real pattern**{A343a}.
b) **complex marker**{A341j} **: letter i**{94a} **symbol.**

{Example:

a) *+.12de+2d 3q"+jx"3"₊" si +.10de+2d* }

{For the semantics of **complex-patterns** see 10.3.4.1.2.e.}

{aa) The modes which are output (input) compatible with a **'complex'** pattern are those specified by ***L compl***, ***L real*** and ***L int*** (by ***ref L compl***).

bb) A value V is converted to a string S using a **'complex'** pattern P as follows:
• if necessary, V is widened to a complex number;
• the element of S controlled by the *"i"* frame of P is *"⊥"*;
• the part of S controlled by the first (second) **'real'** pattern of P is that obtained by converting the first (second) field of V to a string using the first (second) **'real'** pattern of P (10.3.4.3.1.bb);
• if either conversion is unsuccessful, the conversion of V is unsuccessful.

cc) A string is converted to a complex value C suitable for a name N, using a **'complex'** pattern P, as follows:
• the part of the string controlled by the first (second) **'real'** pattern of P is converted to a suitable real number (10.3.4.3.1.cc), which then forms the first (second) field of C;
• if either conversion is unsuccessful, the conversion to C is unsuccessful.}

10.3.4.6. String patterns

10.3.4.6.1. Syntax

a) **NEST string pattern**{A341c} :
**NEST character frame**{A341k} **sequence.**
b) **character marker**{A341k} : **letter a**{94a} **symbol.**

{Example:

a) *p "table of"x 10a* }

{For the semantics of **string-patterns** see 10.3.4.1.2.e.}

{aa) The modes which are output (input) compatible with a **'string'** pattern are those specified by ***char*** and [ ] ***char*** (by ***ref char***, ***ref*** [ ] ***char*** and ***ref string***).

bb) A value V is converted to a string using a **'string'** pattern P as follows:
- if necessary, V is rowed to a string;
- if the length of the string V is equal to the length of the string controlled by P, then V is supplied; otherwise, the conversion is unsuccessful.

cc) A string S is converted to a character or a string suitable for a name N, using a **'string'** pattern, as follows:
Case A: The mode of N is specified by ***ref char***:
- if S does not consist of one character, the conversion is unsuccessful; otherwise, that character is supplied;

Case B: The mode of N is specified by ***ref*** [ ] ***char***:
- if the length of S is not equal to the number of characters referred to by N, the conversion is unsuccessful; otherwise, S is supplied;

Case C: The mode of N is specified by ***ref string***:
- S is supplied.}

10.3.4.7. Bits patterns

10.3.4.7.1. Syntax

a) **NEST bits pattern**{A341c} :
**NEST RADIX frame**{b}, **NEST integral mould**{A342b}.
b) **NEST RADIX frame**{a} : **NEST insertion**{A341d}, **RADIX**{82d,e,f,g},
**unsuppressible suppression**{A341l}, **radix marker**{c}.
c) **radix marker**{b} : **letter r**{94a} **symbol.**

{Examples:

a) *2r6d26sd* b) *2r* }

{For the semantics of **bits-patterns** see 10.3.4.1.2.e.}

{aa) The modes which are output (input) compatible with a **'bits'** pattern are those specified by ***L bits*** (***ref L bits***).

bb) A value V is converted to a string using a **'bits'** pattern P as follows:
• the integer I corresponding to V is determined, using the **operator** ***abs*** (10.2.3.8.i);
If the *"r"* frame of P was yielded by a **radix-two-** (**-four-**, **-eight-**, **-sixteen-**) **-frame**,
then I is converted to a string, controlled by the integral mould of P, containing a binary (quaternary, octal, hexadecimal) representation of I (cf. 10.3.4.2.1.bb);
• if I cannot be represented by such a string, the conversion is unsuccessful.

cc) A string S is converted to a bits value suitable for a name N, using a **'bits'** pattern P, as follows:
• if the *"r"* frame of P was yielded by a **radix-two-** (**-four-**, **-eight-**, **-sixteen-**) **-frame**, then the integer I for which S contains a binary (quaternary, octal, hexadecimal) representation is determined;
• the bits value B corresponding to I is determined, using the **operator** ***bin*** (10.2.3.8.j);
• if the width of B is greater than that of the value to which N refers, the conversion is unsuccessful.}

10.3.4.8. Choice patterns

10.3.4.8.1. Syntax

a) **NEST integral choice pattern**{A341c} : **NEST insertion**{A341d}, **letter c**{94a} **symbol, NEST praglit**{c} **list brief pack, pragment**{92a} **sequence option.**

b) **NEST boolean choice pattern**{A341c} :
**NEST insertion**{A341d}, **boolean marker**{A344b},
**brief begin**{94f} **token, NEST praglit**{c}, **and also**{94f} **token, NEST praglit**{c}, **brief end**{94f} **token, pragment**{92a} **sequence option.**

c) **NEST praglit**{a,b} : **pragment**{92a} **sequence option, NEST literal**{A341i}.

{Examples:

a) *l 20k c ("mon", "tues", "wednes", "thurs", "fri", "satur", "sun")*
b) *b ("", "error")*
c) *"mon"* }

{aa) A value V is output using a picture P whose pattern Q was yielded by an **integral-choice-pattern** C as follows:
• the insertion of Q is staticized (10.3.4.1.1.dd) and performed (10.3.4.1.1.ee);
If the mode of V is specified by ***int***, if V > 0, and if the number of constituent **literals** in the **praglit-list-pack** of C is at least V,

then
- the literal yielded by the V-th **literal** is staticized and performed;

otherwise,
- the event routine corresponding to *on value error* is called;
- if this returns false, V is output using *put* and *undefined* is called;

- the insertion of P is staticized and performed.

bb) A value is input to a name N using a picture P whose pattern Q was yielded by an **integral-choice-pattern** C as follows:
- the insertion of Q is staticized and performed;
- each of the literals yielded by the constituent **literals** of the **praglit-list-pack** of C is staticized and "searched for" (cc) in turn;

If the mode of N is specified by ***ref int*** and the i-th literal is the first one present,
then i is assigned to N;
otherwise,
- the event routine corresponding to *on value error* is called;
- if this returns false, *undefined* is called;

- the insertion of P is staticized and performed.

cc) A literal is "searched for" by reading characters and matching them against successive characters of the literal. If the end of the current line or the logical end of the file is reached, or if a character fails to match, the search is unsuccessful and the current position is returned to where it started from.

dd) A value V is output using a picture P whose pattern Q was yielded by a **boolean-choice-pattern** C as follows:
- the insertion of Q is staticized and performed;

If the mode of V is specified by ***bool***,
then
- if V is true (false), the literal yielded by the first (second) constituent **literal** of C is staticized and performed;

otherwise,
- the event routine corresponding to *on value error* is called;
- if this returns false, V is output using *put* and *undefined* is called;

- the insertion of P is staticized and performed.

ee) A value is input to a name N using a picture P whose pattern Q was yielded by a **boolean-choice-pattern** C as follows:
- the insertion of Q is staticized and performed;
- each of the literals yielded by the constituent **literals** of C is staticized and searched for in turn;

If the mode of N is specified by ***ref bool***, and the first (second) insertion is present,
then true (false) is assigned to N;

otherwise,
- the event routine corresponding to *on value error* is called;
- if this returns false, *undefined* is called;

- the insertion of P is staticized and performed.}

10.3.4.8.2. Semantics

The yield of a **choice-pattern** P is a structured value W whose mode is **'CPATTERN'**, determined as follows:
- let n be the number of constituent **NEST-literals** of the **praglit-list-pack** of P;
- let $S_i$, i = 1, ... , n, be a **NEST-insertion** akin {1.1.3.2.k} to the i-th of those constituent **NEST-literals**;
- the insertion I of P and all of $S_1$, $S_2$, ... , $S_n$ are elaborated collaterally;
- the fields of W, taken in order, are
  - {*i*} the yield of I;
  - {*type*} 1 (2) if P is a **boolean-** (**integral-**) **-choice-pattern**;
  - {*c*} a multiple value whose mode is **'row of INSERTION'**, having a descriptor ((1, n)) and n elements, that selected by (i), i = 1, ... , n, being the yield of $S_i$.

10.3.4.9. Format patterns

10.3.4.9.1. Syntax

a) **NEST format pattern**{A341c} :
**NEST insertion**{A341d}, **letter f**{94a} **symbol**,
**meek FORMAT NEST ENCLOSED clause**{31a,34a},
**pragment**{92a} **sequence option.**

{Example:

a) *f(uir| **(int)**: $ 5d $, **(real)**: $ d.3d $)* }

{A **format-pattern** may be used to provide formats dynamically for use in transput. When a **'format'** pattern is encountered during a call of *get next picture*, it is staticized and its insertion is performed. The first picture of the format returned by the routine of the pattern is supplied as the next picture, and subsequent pictures are taken from that format until it has been exhausted.}

10.3.4.9.2. Semantics

The yield, in an environ E, of a **NEST-format-pattern** P is a structured value whose mode is **'FPATTERN'** and whose fields, taken in order, are
- {*i*} the yield of its **insertion**;
- {*pf*} a routine whose mode is **'procedure yielding FORMAT'**,

composed of a **procedure-yielding-FORMAT-NEST-routine-text** whose **unit** U is a new **unit** akin {1.1.3.2.k} to the **meek-FORMAT-ENCLOSED-clause** of P, together with the environ necessary for U in E.

10.3.4.10. General patterns

10.3.4.10.1. Syntax

a) **NEST general pattern**{A341c} : **NEST insertion**{A341d},
**letter g**{94a} **symbol, NEST width specification**{b} **option.**

b) **NEST width specification**{a} : **brief begin**{94f} **token,**
**meek integral NEST unit**{32d},
**NEST after specification**{c} **option, brief end**{94f} **token,**
**pragment**{92a} **sequence option.**

c) **NEST after specification**{b} :
**and also**{94f} **token, meek integral NEST unit**{32d},
**NEST exponent specification**{d} **option.**

d) **NEST exponent specification**{c} :
**and also**{94f} **token, meek integral NEST unit**{32d).

{Examples:

a) *g • g(-18, 12, -3)* b) *-18, 12, -3*
c) *, 12, -3* d) *, -3* }

{aa) A value V is output using a picture P whose pattern Q was yielded by a **general-pattern** G as follows:
- P is staticized;
- the insertion of Q is performed;

If Q is not parametrized (i.e., G contains no **width-specification**),
then V is output using *put*;
otherwise, if the mode of V is specified by ***L int*** or ***L real***,
then
- if Q contains one (two, three) parameter(s), V is converted to a string using *whole* (*fixed*, *float*);
- the string is written using *put*;

otherwise,
- the event routine corresponding to *on value error* is called;
- if this returns false, V is output using *put*, and *undefined* is called;
- the insertion of P is performed.

bb) A value is input to a name N using a picture P whose pattern is a **'general'** pattern as follows:
- P is staticized;
- (any parameters are ignored and) the value is input to N using *get*.}

10.3.4.10.2. Semantics

The yield, in an environ E, of a **NEST-general-pattern** P is a structured value whose mode is **'GPATTERN'** and whose fields, taken in order, are
- {*i*} the yield of the **insertion** of P;

• {*spec*} a multiple value W whose mode is '**row of procedure yielding integral**', having a descriptor ((1, n)), where n is the number of constituent **meek-integral-units** of the **width-specification-option** of P, and n elements determined as follows:
For i = 1, ... , n,

• the i-th element of W is a routine, whose mode is '**procedure yielding integral**', composed of a **procedure-yielding-integral-NEST-routine-text** whose **unit** U is a new **unit** akin {1.1.3.2.k} to the i-th of those **meek-integral-units**, together with the environ necessary for U in E.

10.3.5. Formatted transput

a)
```
mode format = struct (flex [1 : 0] piece F);
mode ? piece = struct (int cp ¢ pointer to current collection ¢,
    count ¢ number of times piece is to be repeated ¢,
    bp ¢ back pointer ¢,
    flex [1 : 0] collection c);
mode ? collection = union (picture, collitem);
mode ? collitem = struct (insertion i1,
    proc int rep ¢ replicator ¢,
    int p ¢ pointer to another piece ¢, insertion i2);
mode ? insertion = flex [1 : 0] struct (proc int rep ¢ replicator ¢,
    union (string, char) sa);
mode ? picture =
    struct (union (pattern, cpattern, fpattern, gpattern, void) p, insertion i);
mode ? pattern = struct (int type ¢ of pattern ¢,
    flex [1 : 0] frame frames);
mode ? frame = struct (insertion i,
    proc int rep ¢ replicator ¢,
    bool supp ¢ true if suppressed ¢,
    char marker);
mode ? cpattern = struct (insertion i,
    int type ¢ boolean or integral ¢,
    flex [1 : 0] insertion c);
mode ? fpattern = struct (insertion i, proc format pf);
mode ? gpattern = struct (insertion i, flex [1 : 0] proc int spec);
```

b)
```
proc ? get next picture = (ref file f, bool read, ref picture picture) void :
    begin
    bool picture found := false, format ended := false;
    while ¬ picture found
    do if forp of f = 0 then
        if format ended
        then undefined
        elif ¬ (format mended of f) (f)
        then ref int (forp of f) := 1;
            cp of (F of format of f) [1] := 1;
```

```
        count of (F of format of f) [1] := 1
    else format ended := true
    fi
else
    ref int forp = forp of f;
    ref flex [ ] piece aleph = F of format of f;
    case (c of aleph [forp]) [cp of aleph [forp] ] in
        (collitem cl):
            ([1 : upb (i1 of cl)] insert si;
            bp of aleph [p of cl] := forp; forp := skip;
            (staticize insertion (i1 of cl, si),
                count of aleph [p of cl] := rep of cl);
            (aleph :≠: F of format of f | undefined);
            (read | get insertion (f, si) | put insertion (f, si));
            cp of aleph [p of cl] := 0;
            forp := p of cl),
        (picture pict): (picture found := true; picture := pict)
    esac;
    while
        (forp ≠ 0 | cp of aleph [forp] = upb c of aleph [forp] | false)
    do if (count of aleph [forp] −:= 1) ≤ 0
    then
        if (forp := bp of aleph [forp]) ≠ 0
        then
            insertion extra =
                case (c of aleph [forp]) [cp of aleph [forp] ] in
                (collitem cl):
                    (bp of aleph [p of cl] := 0; i2 of cl),
                (picture pict):
                    case p of pict in
                    (fpattern fpatt):
                        (int k := forp;
                        while bp of aleph [k] ≠ forp do k +:= 1 od;
                        aleph := aleph [ : k − 1];
                        i of pict)
                    esac
                esac;
            int m = upb i of picture, n = upb extra;
            [1 : m + n] struct (proc int rep, union (string, char) sa) c;
            c [1 : m] := i of picture; c [m + 1 : m + n] := extra;
            i of picture := c
        fi
    else cp of aleph [forp] := 0
    fi od;
    (forp ≠ 0 | cp of aleph [forp] +:= 1)
fi od
end ;
```

c)
```
mode ? sinsert = struct (int rep, union (string, char) sa) ;
```

d)
```
proc ? staticize insertion = (insertion ins, ref [ ] sinsert sins) void :
      ¢ calls collaterally all the replicators in 'ins' ¢
   if upb ins = 1
   then
      rep of sins [1] := rep of ins [1];
      sa of sins [1] := sa of ins [1]
   elif upb ins > 1
   then (staticize insertion (ins [1], sins [1]),
         staticize insertion (ins [2 : ], sins [2 : ]))
   fi ;
```

e)
```
mode ? sframe = struct (flex [1 : 0] sinsert si, int rep, bool supp,
   char marker) ;
```

f)
```
proc ? staticize frames =
   ([ ] frame frames, ref [ ] sframe sframes) void :
      ¢ calls collaterally all the replicators in 'frames' ¢
   if upb frames = 1
   then
      [1 : upb (i of frames [1])] sinsert si;
      (staticize insertion (i of frames [1], si),
         rep of sframes [1] := rep of frames [1]);
      si of sframes [1] := si;
      supp of sframes [1] := supp of frames [1];
      marker of sframes [1] := marker of frames [1]
   elif upb frames > 1
   then (staticize frames (frames [1], sframes [1]),
         staticize frames (frames [2 : ], sframes [2 : ]))
   fi ;
```

g)
```
proc ? put insertion = (ref file f, [ ] sinsert si) void :
   begin set write mood (f);
      for k to upb si
      do
         case sa of si [k] in
            (char a): alignment (f, rep of si [k], a, false) ,
            (string s):
               to rep of si [k]
               do
                  for i to upb s
                  do check pos (f); put char (f, s [i]) od
               od
         esac
      od
   end ;
```

h)
```
proc ? get insertion = (ref file f, [ ] sinsert si) void :
    begin set read mood (f);
        for k to upb si
        do
            case sa of si [k] in
                (char a): alignment (f, rep of si [k], a, true),
                (string s):
                    (char c;
                    to rep of si [k]
                    do
                        for i to upb s
                        do check pos (f); get char (f, c);
                            (c ≠ s [i]
                            | (¬ (char error mended of f) (f, c := s [i])
                              | undefined);
                            set read mood (f))
                        od
                    od)
            esac
        od
    end ;
```

i)
```
proc ? alignment = (ref file f, int r, char a, bool read) void :
    if a = "x" then to r do space (f) od
    elif a = "y" then to r do backspace (f) od
    elif a = "l" then to r do newline (f) od
    elif a = "p" then to r do newpage (f) od
    elif a = "k" then set char number (f, r)
    elif a = "q"
    then to r
        do
            if read
            then char c; check pos (f); get char (f, c);
                (c ≠ blank
                | (¬ (char error mended of f) (f, c := blank)
                  | undefined); set read mood (f))
            else check pos (f); put char (f, blank)
            fi
        od
    fi ;
```

j)
```
proc ? do fpattern = (ref file f, fpattern fpattern, bool read) void :
    begin format pf;
        [1 : upb (i of fpattern)] sinsert si;
        (staticize insertion (i of fpattern, si),
            pf := pf of fpattern);
        (read | get insertion (f, si) | put insertion (f, si));
```

```
        ref int forp = forp of f;
        ref flex [ ] piece aleph = F of format of f;
        int m = upb aleph, n = upb (F of pf);
        [1 : m + n] piece c; c [1 : m] := aleph;
        c [m + 1 : m + n] := F of pf;
        aleph := c; bp of aleph [m + 1] := forp;
        forp := m + 1; cp of aleph [forp] := 0;
        count of aleph [forp] := 1;
        for i from m + 1 to m + n
        do
            for j to upb c of aleph [i]
            do
                case (c of aleph [i]) [j] in
                    (collitem cl):
                        (c of aleph [i]) [j] :=
                            collitem (i1 of cl, rep of cl, p of cl + m, i2 of cl)
                esac
            od
        od
    end;
```

k)

```
    proc ? associate format = (ref file f, format format) void :
        begin
            format of f :=
                c a newly created name which is made to refer to the yield
                of an actual-format-declarer and whose scope is equal to
                the scope of the value yielded by 'format' c
                    := format;
            forp of f := heap int := 1;
            cp of (F of format of f) [1] := 1;
            count of (F of format of f) [1] := 1;
            bp of (F of format of f) [1] := 0
        end;
```

10.3.5.1. Formatted output

a)

```
    proc putf = (ref file f, [ ] union (outtype, format) x) void :
        if opened of f then
        for k to upb x
        do case set write mood (f); set char mood (f); x [k] in
            (format format): associate format (f, format),
            (outtype ot):
            begin int j := 0;
                picture picture, [ ] simplout y = straightout ot;
                while (j +:= 1) ≤ upb y
                do bool incomp := false;
                    get next picture (f, false, picture);
```

```
set write mood (f);
[1 : upb (i of picture)] sinsert sinsert;
case p of picture in

(pattern pattern):
begin int rep, sfp := 1;
    [1 : upb (frames of pattern)] sframe sframes;
    (staticize frames (frames of pattern, sframes),
        staticize insertion (i of picture, sinsert));
    string s;

    op ? = (string s) bool :
        ¢ true if the next marker is one of the elements of
            's' and false otherwise ¢
        if sfp > upb sframes
        then false
        else sframe sf = sframes [sfp];
            rep := rep of sf;
            if char in string (marker of sf, loc int, s)
            then sfp +:= 1; true
            else false
            fi
        fi;

    op ? = (char c) bool : ? string (c);

    proc int pattern = (ref bool sign mould) int :
        (int l := 0;
        while ? "zuv" do (rep ≥ 0 | l +:= rep) od;
        sign mould := ? "+-";
        while ? "zd" do (rep ≥ 0 | l +:= rep) od; l);

    ↓proc edit L int = (L int i) void :
        (bool sign mould; int l := int pattern (sign mould);
        string t = subwhole (abs i, l);
        if char in string (errorchar, loc int, t) ∨ l = 0
            ∨ ¬ sign mould ∧ i < L 0
        then incomp := true
        else t plusto s;
            (l - upb t) × "0" plusto s;
            (sign mould | (i < L 0 | "-" | "+") plusto s)
        fi)↓;

    ↓proc edit L real = (L real r) void :
        (int b := 0, a := 0, e := 0, exp := 0, L real y := abs r,
        bool sign1, string point := "";
        b := int pattern (sign1);
        (? "." | a := int pattern (loc bool); point := ".");
        if ? "e"
        then L standardize (y, b, a, exp);
```

```
        edit int (exp);
        "10" plusto s
    fi;
    string t = subfixed (y, b + a + (a ≠ 0 | 1 | 0), a);
    if char in string (errorchar, loc int, t) ∨ a + b = 0
        ∨ ¬ sign1 ∧ r < L 0
    then incomp := true
    else t [ : b] + point + t [b + 2: ] plusto s;
        (b + a + (a ≠ 0 | 1 | 0) - upb t) × "0" plusto s;
        (sign1 | (r < L 0 | "-" | "+") plusto s)
    fi)};

{proc edit L compl = (L compl z) void :
    (while ¬ ? "i" do sfp +:= 1 od; edit L real (im z);
    "⊥" plusto s; sfp := 1; edit L real (re z))};

{proc edit L bits = (L bits lb, int radix) void :
    (L int n := abs lb; ? "r"; int l := int pattern (loc bool);
    while dig char (S (n mod K radix)) plusto s;
        n ÷:= K radix; n ≠ L 0
    do skip od;
    if upb s ≤ l
    then (l - upb s) × "0" plusto s
    else incomp := true
    fi)};

proc charcount = int : (int l := 0;
    while ? "a" do (rep ≥ 0 | l +:= rep) od; l);

case type of pattern in
¢ integral ¢
        (y [j] |
        {(L int i): edit L int (i)}
        | incomp := true),
¢ real ¢
        (y [j] |
        {(L real r): edit L real (r)},
        {(L int i): edit L real (i)}
        | incomp := true),
¢ boolean ¢
        (y [j] |
        (bool b): s := (b | flip | flop)
        | incomp := true),
¢ complex ¢
        (y [j] |
        {(L compl z): edit L compl (z)},
        {(L real r): edit L compl (r)},
        {(L int i): edit L compl (i)}
        | incomp := true),
```

```
¢ string ¢
        (y [j] |
        (char c): (charcount = 1 | s := c | incomp := true),
        ([ ] char t):
            (charcount = upb t - lwb t + 1
            | s := t [@ 1]
            | incomp := true)
        | incomp := true)
    out
¢ bits ¢
        (y [j] |
        {(L bits lb): edit L bits (lb, type of pattern - 4)}
        | incomp := true)
    esac;
    if ¬ incomp
    then edit string (f, s, sframes)
    fi
end,

(cpattern choice):
begin
    [1 : upb (i of choice)] sinsert si;
    staticize insertion (i of choice, si);
    put insertion (f, si);
    int l =
        case type of choice in
    ¢ boolean ¢
            (y [j] |
            (bool b): (b | 1 | 2)
            | incomp := true; skip),
    ¢ integral ¢
            (y [j] |
            (int i): i
            | incomp := true; skip)
        esac;
    if ¬ incomp
    then
        if l > upb (c of choice) ∨ l ≤ 0
        then incomp := true
        else
            [1 : upb ((c of choice) [l])] sinsert ci;
            staticize insertion ((c of choice) [l], ci);
            put insertion (f, ci)
        fi
    fi;
    staticize insertion (i of picture, sinsert)
end,
```

```
            (fpattern fpattern):
            begin
                do fpattern (f, fpattern, false);
                for i to upb sinsert do sinsert [i] := (0, "") od;
                j -:= 1
            end ,

            (gpattern gpattern):
            begin
                [1 : upb (i of gpattern)] sinsert si;
                [ ] proc int spec = spec of gpattern; int n = upb spec;
                [1 : n] int s;
                (staticize insertion (i of gpattern, si),
                    staticize insertion (i of picture, sinsert),
                    s := (n | spec [1], (spec [1], spec [2]),
                        (spec [1], spec [2], spec [3]) | () ));
                put insertion (f, si);
                if n = 0 then put (f, y [j])
                else
                    number yj =
                        (y [j] | ¢(L int i): i¢, ¢(L real r): r¢
                        | incomp := true; skip);
                    if ¬ incomp
                    then case n in
                        put (f, whole (yj, s [1])) ,
                        put (f, fixed (yj, s [1], s [2])) ,
                        put (f, float (yj, s [1], s [2], s [3]))
                        esac
                    fi
                fi
            end ,

            (void):
                (j -:= 1; staticize insertion (i of picture, sinsert))

            esac;
            if incomp
            then set write mood (f);
                (¬ (value error mended of f) (f) | put (f, y [j]);
                    undefined)
            fi;
            put insertion (f, sinsert)
        od
    end
esac od
else undefined
fi ;
```

b)

```
proc ? edit string = (ref file f, string s, [ ] sframe sf) void :
  begin bool supp, zs := true, signput := false, again, int j := 0, sign;
    proc copy = (char c) void :
      (¬ supp | check pos (f); put char (f, c));
    for k to upb sf
    do sframe sfk = sf [k]; supp := supp of sfk;
      put insertion (f, si of sfk);
      to rep of sfk
      do again := true;
        while again
        do j +:= 1; again := false;
          char sj = s [j], marker = marker of sfk;
          if marker = "d"
          then copy (sj); zs := true
          elif marker = "z" then
            (sj = "0" | copy ((zs | "±" | sj))
            |: sj = "+" | again := true
            | zs := false; copy (sj))
          elif marker = "u" ∨ marker = "v" then
            (sj = "+" | sign := 1; again := true
            |: sj = "−" | sign := 2; again := true
            |: sj = "0" | copy ((zs | "±" | sj))
            | (¬ signput |
              copy ((sign | (marker = "u" | "+" | "±"), "−"));
              signput := true);
              copy (sj); zs := false)
          elif marker = "+" then
            (sj = "+" ∨ sj = "−" | copy (sj)
            | (¬ signput | copy ((sign | "+", "−")));
              j −:= 1)
          elif marker = "−" then
            (sj = "+" | copy ("±")
            |: sj = "−" | copy (sj)
            | (¬ signput | copy ((sign | "±", "−")));
              j −:= 1)
          elif marker = "." then
            copy (".")
          elif marker = "e" ∨ marker = "i"
            ∨ marker = "a" ∨ marker = "b"
          then copy (sj); zs := true; signput := false
          elif marker = "r"
          then j −:= 1
          fi
        od
      od
    od
  end ;
```

10.3.5.2. Formatted input

```
a)  proc getf = (ref file f, [ ] union (intype, format) x) void :
      if opened of f then
      for k to upb x
      do case set read mood (f); set char mood (f); x[k] in
         (format format): associate format (f, format),
         (intype it):
         begin int j := 0;
            picture picture, [ ] simplin y = straightin it;
            while (j +:= 1) ≤ upb y
            do bool incomp := false;
               get next picture (f, true, picture); set read mood (f);
               [1 : upb (i of picture)] sinsert sinsert;
               case p of picture in

               (pattern pattern):
               begin
                  [1 : upb (frames of pattern)] sframe sframes;
                  (staticize frames (frames of pattern, sframes),
                     staticize insertion (i of picture, sinsert));
                  string s;
                  int radix =
                     (type of pattern ≥ 6 | type of pattern - 4 | 10);
                  indit string (f, s, sframes, radix);
                  case type of pattern in
               ¢ integral ¢
                     (y [j] |
                     ↓(ref L int ii):
                        incomp := ¬ string to L int (s, 10, ii)↑
                     | incomp := true),
               ¢ real ¢
                     (y [j] |
                     ↓(ref L real rr):
                        incomp := ¬ string to L real (s, rr)↑
                     | incomp := true),
               ¢ boolean ¢
                     (y [j] |
                     (ref bool bb): bb := s = flip
                     | incomp := true),
               ¢ complex ¢
                     (y [j] |
                     ↓(ref L compl zz):
                        (int i, bool b1, b2; char in string ("⊥", i, s);
                        b1 := string to L real (s [ : i - 1], re of zz);
                        b2 := string to L real (s [i + 1 : ], im of zz);
                        incomp := ¬ (b1 ∧ b2))↑
                     | incomp := true),
```

```
¢ string ¢
        (y [j] |
        (ref char cc):
            (upb s = 1 | cc := s [1] | incomp := true),
        (ref [ ] char ss):
            (upb ss - lwb ss + 1 = upb s | ss [@ 1] := s
            | incomp := true),
        (ref string ss): ss := s
        | incomp := true)
    out
¢ bits ¢
        (y [j] |
        ǂ(ref L bits lb):
            if L int i; string to L int (s, radix, i)
            then lb := bin i
            else incomp := true
            fi ǂ
        | incomp := true)
    esac
end,

(cpattern choice):
begin
    [1 : upb (i of choice)] sinsert si;
    staticize insertion (i of choice, si);
    get insertion (f, si);
    int c = c of cpos of f, char kk;
    int k := 0, bool found := false;
    while k < upb (c of choice) ∧ ¬ found
    do k +:= 1;
        [1 : upb ((c of choice) [k])] sinsert si;
        bool bool := true;
        staticize insertion ((c of choice) [k], si);
        string s;
        for i to upb si
        do s plusab
            (sa of si [i] | (string ss): ss) × rep of si [i]
        od;
        for jj to upb s
        while bool := bool ∧ ¬ line ended (f)
            ∧ ¬ logical file ended (f)
        do get char (f, kk); bool := kk = s [jj] od;
        (¬ (found := bool) | set char number (f, c))
    od;
    if ¬ found then incomp := true
    else
        case type of choice in
```

```
            ¢ boolean ¢
                    (y [j] |
                    (ref bool b): b := k = 1
                    | incomp := true),
            ¢ integral ¢
                    (y [j] |
                    (ref int i): i := k
                    | incomp := true)
                esac
            fi;
            staticize insertion (i of picture, sinsert)
        end,

        (fpattern fpattern):
        begin do fpattern (f, fpattern, true);
            for i to upb sinsert do sinsert [i] := (0, "") od;
            j −:= 1
        end,

        (gpattern gpattern):
            ([1 : upb (i of gpattern)] sinsert si;
            (staticize insertion (i of gpattern, si),
                staticize insertion (i of picture, sinsert));
            get insertion (f, si);
            get (f, y [j])),

        (void):
            (j −:= 1; staticize insertion (i of picture, sinsert))

        esac;
        if incomp
        then set read mood (f);
            (¬ (value error mended of f) (f) | undefined)
        fi;
        get insertion (f, sinsert)
    od
  end
esac od
else undefined
fi;
```

b)

```
proc ? indit string = (ref file f, ref string s, [ ] sframe sf, int radix) void :
  begin
    bool supp, zs := true, sign found := false, space found := false,
        no sign := false, int sp := 1, rep;
    prio ! = 8;
```

```
op ! = (string s, char c) char :
        ¢ expects a character contained in 's'; if the character
        read is not in 's', the event routine corresponding to 'on
        char error' is called with the suggestion 'c' ¢
    if char k; check pos (f); get char (f, k);
        char in string (k, loc int, s)
    then k
    else char sugg := c;
        if (char error mended of f) (f, sugg) then
            (char in string (sugg, loc int, s) | sugg | undefined; c)
        else undefined; c
        fi;
        set read mood (f)
    fi;
op ! = (char s, c) char : string (s) ! c;
[ ] char good digits = "0123456789abcdef" [ : radix ];
s := "+";
for k to upb sf
do sframe sfk = sf [k]; supp := supp of sfk;
    get insertion (f, si of sfk);
    to rep of sfk
    do char marker = marker of sfk;
        if marker = "d" then
            s plusab (supp | "0" | good digits ! "0"); zs := true
        elif marker = "z" then
            s plusab (supp | "0"
                | char c = ((zs | "±" | "") + good digits) ! "0";
                    (c ≠ "±" | zs := false); c)
        elif marker = "u" ∨ marker = "+" then
            if sign found
            then zs := false; s plusab ("0123456789" ! "0")
            else char c = ("+-" + (marker = "u" | "±" | "")) ! "+";
                (c = "+" ∨ c = "-" | sign found := true; s [sp] := c)
            fi
        elif marker = "v" ∨ marker = "-" then
            if sign found
            then zs := false; s plusab ("0123456789" ! "0")
            elif char c; space found
            then c := "+-±0123456789" ! "+";
                (c = "+" ∨ c = "-" | sign found := true; s [sp] := c
                |: c ≠ "±" | zs := false; sign found := true; s plusab c)
            else c := "+-±" ! "+";
                (c = "+" ∨ c = "-" | sign found := true; s [sp] := c
                | space found := true)
            fi
        elif marker = "." then
            s plusab (supp | "." | "." ! ".")
```

```
            elif marker = "e" then
                s plusab (supp | "10" | "10\e" ! "10"; "10");sign found := false;
                zs := true; s plusab "+"; sp := upb s
            elif marker = "i" then
                s plusab (supp | "⊥" | "i⊥" ! "⊥"; "⊥");
                sign found := false; zs := true; s plusab "+"; sp := upb s
            elif marker = "b" then
                s plusab (flip + flop) ! flop; no sign := true
            elif marker = "a" then
                s plusab (supp | "." | char c; check pos (f); get char (f, c);
                c);
                no sign := true
            elif marker = "r"
            then skip
            fi
        od
    od;
    if no sign then s := s [2 : ] fi
end;
```

10.3.6. Binary transput

{In binary transput, the values obtained by straightening the elements of a data list (cf. 10.3.3) are transput, via the specified file, one after the other. The manner in which such a value is stored in the book is defined only to the extent that a value of mode M (being some mode from which that specified by ***simplout*** is united) output at a given position may subsequently be re-input from that same position to a name of mode **'reference to** M'. Note that, during input to the name referring to a multiple value, the number of elements read will be the existing number of elements referred to by that name.

The current position is advanced after each value by a suitable amount and, at the end of each line or page, the appropriate event routine is called, and next, if this returns false, the next good character position of the book is found (cf. 10.3.3).

For binary output, *put bin* (10.3.6.1.a) and *write bin* (10.5.1.h) may be used and, for binary input, *get bin* (10.3.6.2.a) and *read bin* (10.5.1.i).}

a) ***proc*** ? *to bin* = (***ref file*** *f*, ***simplout*** *x*) [ ] ***char*** :
***c*** *a value of mode 'row of character' whose lower bound is one and whose upper bound depends on the value of 'book* ***of*** *f' and on the mode and the value of 'x'; furthermore, x = from bin (f, x, to bin (f, x))* ***c*** ;

b) ***proc*** ? *from bin* = (***ref file*** *f*, ***simplout*** *y*, [ ] ***char*** *c*) ***simplout*** :
***c*** *a value, if one exists, of the mode of the value yielded by 'y', such that c = to bin (f, from bin (f, y, c))* ***c*** ;

10.3.6.1. Binary output

```
a)  proc put bin = (ref file f, [ ] outtype ot) void :
      if opened of f then
        set bin mood (f); set write mood (f);
        for k to upb ot
        do [ ] simplout y = straightout ot [k];
          for j to upb y
          do [ ] char bin = to bin (f, y [j]);
            for i to upb bin
            do next pos (f);
              set bin mood (f);
              ref pos cpos = cpos of f, lpos = lpos of book of f;
              case text of f in
              (flextext t2):
                t2 [p of cpos] [l of cpos] [c of cpos] := bin [i]
              esac;
              c of cpos +:= 1;
              if cpos beyond lpos then lpos := cpos
              elif ¬ set possible (f)
                ^ pos (p of lpos, l of lpos, 1) beyond cpos
              then lpos := cpos;
                (compressible (f) |
                  c the size of the line and page containing the
                    logical end of the book and of all
                    subsequent lines and pages may be
                    increased c)
              fi
            od
          od
        od
      else undefined
      fi ;
```

10.3.6.2. Binary input

```
a)  proc get bin = (ref file f, [ ] intype it) void :
      if opened of f then
        set bin mood (f); set read mood (f);
        for k to upb it
        do [ ] simplin y = straightin it [k];
          for j to upb y
          do
            simplout yj = case y [j] in
              ł(ref L int i): i ł , ł(ref L real r): r ł ,
              ł(ref L compl z): z ł , (ref bool b): b ,
              ł(ref L bits lb): lb ł , (ref char c): c , (ref [ ] char s): s ,
              (ref string ss): ss esac;
```

```
        [1 : upb (to bin (f, yj))] char bin;
        for i to upb bin
        do next pos (f); set bin mood (f);
            ref pos cpos = cpos of f;
            bin [i] :=
                case text of f in
                (flextext t2):
                    t2 [p of cpos] [l of cpos] [c of cpos]
                esac;
            c of cpos +:= 1
        od;
        case y [j] in
            ¢(ref L int ii): (from bin (f, ii, bin) | (L int i): ii := i)¢,
            ¢(ref L real rr):
                (from bin (f, rr, bin) | (L real r): rr := r)¢,
            ¢(ref L compl zz):
                (from bin (f, zz, bin) | (L compl z): zz := z)¢,
            (ref bool bb): (from bin (f, bb, bin) | (bool b): bb := b),
            ¢(ref L bits lb):
                (from bin (f, lb, bin) | (L bits b): lb := b)¢,
            (ref char cc): (from bin (f, cc, bin) | (char c): cc := c),
            (ref [ ] char ss):
                (from bin (f, ss, bin) | ([ ] char s): ss := s),
            (ref string ssss):
                (from bin (f, ssss, bin) | ([ ] char s): ssss := s)
        esac
    od
od
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.}

{

}

## 10.4. The system prelude and task list

### 10.4.1. The system prelude

The representation of the **system-prelude** is obtained from the following form, to which may be added further forms not defined in this Report. {The syntax of **program-texts** ensures that a **declaration** contained in the **system-prelude** may not contradict any **declaration** contained in the **standard-prelude**. It is intended that the further forms should contain declarations that are needed for the correct operation of any **system-tasks** that may be added (by the implementer, as provided in 10.1.2.d).}

a) ***sema*** ? *gremlins* = (***sema*** *s*; *F* ***of*** *s* := ***PRIM int*** := *0*; *s*);

### 10.4.2. The system task list

The representation of the {first} constituent **system-task** of the **system-task-list** is obtained from the following form. The other **system-tasks**, if any, are not defined by this Report {but may be defined by the implementer in order to account for the particular features of his operating environment, especially in so far as they interact with the running of **particular-programs** (see, e.g., 10.3.1.1.dd)}.

a) ***do down*** *gremlins*; *undefined*; ***up*** *bfileprotect* ***od***

{The intention is that this call of *undefined*, which is released by an ***up*** *gremlins* whenever a book is closed, may reorganize the chain of backfiles and the chain of locked backfiles, such as by removing the book if it is not to be available for further opening, or by inserting it into the chain of backfiles several times over if it is to be permitted for several **particular-programs** to read it simultaneously. Note that, when an ***up*** *gremlins* is given, *bfileprotect* is always down and remains so until such reorganization has taken place.}

{From ghoulies and ghosties and long-leggety beasties and things that go bump in the night,
Good Lord, deliver us!
Ancient Cornish litany}

## 10.5. The particular preludes and postludes

### 10.5.1. The particular preludes

The representation of the **particular-prelude** of each **user-task** is obtained from the following forms, to which may be added such other forms as may be needed for the proper functioning of the facilities defined in the constituent **library-prelude** of the **program-text** {, e.g., **declarations** and calls of *open* for additional standard files}. However, for each **QUALITY-new-new-PROPS1-LAYER2-defining-indicator-with-TAX** contained

in such an additional form, the predicate **'where QUALITY TAX independent PROPS1'** {7.1.1.a,c} must hold {i.e., no **declaration** contained in the **standard-prelude** may be contradicted}.

a) ***L int*** *L last random* := ***round*** *(L max int / **L** 2);*

b) ***proc*** *L random* = ***L real*** : *L next random (L last random);*

c) ***file*** *stand in, stand out, stand back;*
*open (stand in, "", stand in channel);*
*open (stand out, "", stand out channel);*
*open (stand back, "", stand back channel);*

d) ***proc*** *print* = *([ ]* ***union (outtype, proc (ref file) void)*** *x)* ***void*** :
*put (stand out, x),*
***proc*** *write* = *([ ]* ***union (outtype, proc (ref file) void)*** *x)* ***void*** :
*put (stand out, x);*

e) ***proc*** *read* = *([ ]* ***union (intype, proc (ref file) void)*** *x)* ***void*** :
*get (stand in, x);*

f) ***proc*** *printf* = *([ ]* ***union (outtype, format)*** *x)* ***void*** : *putf (stand out, x),*
***proc*** *writef* = *([ ]* ***union (outtype, format)*** *x)* ***void*** : *putf (stand out, x);*

g) ***proc*** *readf* = *([ ]* ***union (intype, format)*** *x)* ***void*** : *getf (stand in, x);*

h) ***proc*** *write bin* = *([ ]* ***outtype*** *x)* ***void*** : *put bin (stand back, x);*

i) ***proc*** *read bin* = *([ ]* ***intype*** *x)* ***void*** : *get bin (stand back, x);*

### 10.5.2. The particular postludes

The representation of the **particular-postlude** of each **user-task** is obtained from the following form, to which may be added such other forms as may be needed for the proper functioning of the facilities defined in the constituent **library-prelude** of the **program-text** {, e.g., **calls** of *lock* for additional standard files}.

a) *stop: lock (stand in); lock (stand out); lock (stand back)*

## 11. Examples

### 11.1. Complex square root

***proc*** *compsqrt* = *(****compl*** *z)* ***compl*** :
*¢ the square root whose real part is nonnegative of the complex number 'z' ¢*
***begin real*** *x* = ***re*** *z, y* = ***im*** *z;* ***real*** *rp* = *sqrt ((****abs*** *x* + *sqrt (x ↑ 2* + *y ↑ 2)) / 2);*
***real*** *ip* = *(rp* = *0 | 0 | y / (2* × *rp));*

```
        if x ≥ 0 then rp ⊥ ip else abs ip ⊥ (y ≥ 0 | rp | - rp) fi
    end
```

**Calls** {5.4.3} using *compsqrt*:
*compsqrt (w)*
*compsqrt (-3.14)*
*compsqrt (-1)*

## 11.2. Innerproduct 1

```
proc innerproduct 1 = (int n, proc (int) real x, y) real :
        ¢ the innerproduct of two vectors, each with 'n' compponents, x (i),
        y (i), i = 1, ... , n, where 'x' and 'y' are arbitrary mappings from
        integer to real number ¢
    begin long real s := long 0;
        for i to n do s +:= leng x (i) × leng y (i) od;
        shorten s
    end
```

**Real-calls** using *innerproduct 1*:
*innerproduct 1 (m, (int j) real : x1 [j], (int j) real : y1 [j])*
*innerproduct 1 (n, nsin, ncos)*

## 11.3. Innerproduct 2

```
proc innerproduct 2 = (ref [ ] real a, b) real :
    if upb a - lwb a = upb b - lwb b
    then ¢ the innerproduct of two vectors 'a' and 'b' with equal numbers
        of elements ¢
        long real s := long 0;
        ref [ ] real a1 = a [@ 1], b1 = b [@ 1];
            ¢ note that the bounds of 'a [@ 1]' are [1 : upb a - lwb a + 1] ¢
        for i to upb a1 do s +:= leng a1 [i] × leng b1 [i] od;
        shorten s
    fi
```

**Real-calls** using *innerproduct 2*:
*innerproduct 2 (x1, y1)*
*innerproduct 2 (y2 [2, ], y2 [ ,3])*

## 11.4. Largest element

```
proc absmax = (ref [,] real a, ¢ result ¢ ref real y,
    ¢ subscripts ¢ ref int i, k) void :
        ¢ 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' ¢
```

```
begin y := - 1;
    for p from 1 lwb a to 1 upb a
    do
        for q from 2 lwb a to 2 upb a
        do
            if abs a [p, q] > y then y := abs a [i := p, k := q] fi
        od
    od
end
```

**Calls** using *absmax*:
*absmax (x2, x, i, j)*
*absmax (x2, x, **loc int**, **loc int**)*

## 11.5. Euler summation

```
proc euler = (proc (int) real f, real eps, int tim) real :
        ¢ the sum for 'i' from 1 to infinity of 'f (i)', computed by means of
        a suitably refined Euler transformation. The summation is
        terminated when the absolute values of the terms of the
        transformed series are found to be less than 'eps' 'tim' times in
        succession. This transformation is particularly efficient in the
        case of a slowly convergent or divergent alternating series ¢
    begin int n := 1, t; real mn, ds := eps; [1 : 16] real m;
        real sum := (m [1] := f (1)) / 2;
        for i from 2 while (t := (abs ds < eps | t + 1 | 1)) ≤ tim
        do mn := f (i);
            for k to n do mn := ((ds := mn) + m [k]) / 2; m [k] := ds od;
            sum +:= (ds := (abs mn < abs m [n] ∧ n < 16 | n +:= 1; m [n] := mn;
                mn / 2 | mn))
        od;
        sum
    end
```

A **call** using *euler*:
*euler ((int i) real : (odd i | - 1 / i | 1 / i), $1_{10}$-5, 2)*

## 11.6. The norm of a vector

```
proc norm = (ref [ ] real a) real :
        ¢ the euclidean norm of the vector 'a' ¢
    (long real s := long 0;
        for k from lwb a to upb a do s +:= leng a [k] ↑ 2 od;
        shorten long sqrt (s))
```

For a use of *norm* in a **call**, see 11.7.

11.7. Determinant of a matrix

```
proc det = (ref [,] real x, ref [ ] int p) real :
    if ref [,] real a = x [@ 1, @ 1];
        1 upb a = 2 upb a ∧ 1 upb a = upb p - lwb p + 1
    then int n = 1 upb a;
            ¢ the determinant of the square matrix 'a' of order 'n' by the
            method of Crout with row interchanges: 'a' is replaced by its
            triangular decomposition, l × u, with all u [k, k] = 1. The
            vector 'p' gives as output the pivotal row indices; the k-th
            pivot is chosen in the k-th column of 'l' such that
            abs l [i, k] / row norm is maximal ¢
        [1 : n] real v; real d := 1, s, pivot;
        for i to n do v [i] := norm (a [i, ]) od;
        for k to n
        do int k1 = k - 1; ref int pk = p [@ 1] [k]; real r := -1;
            ref [,] real al = a [ , 1 : k1 ], au = a [1 : k1, ];
            ref [ ] real ak = a [k, ], ka = a [ , k], alk = al [k, ], kau = au [ , k];
            for i from k to n
            do ref real aik = ka [i];
                if (s := abs (aik -:= innerproduct 2 (al [i, ], kau)) / v [i]) > r
                then r := s; pk := i
                fi
            od;
            v [pk] := v [k]; pivot := ka [pk]; ref [ ] real apk = a [pk, ];
            for j to n
            do ref real akj = ak [j], apkj = apk [j];
                r := akj;
                akj := if j ≤ k then apkj
                    else (apkj - innerproduct 2 (alk, au [ , j])) / pivot fi;
                if pk ≠ k then apkj := - r fi
            od;
            d ×:= pivot
        od;
        d
    fi
```

A **call** using *det*:

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

11.8. Greatest common divisor

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

A **call** using *gcd*:

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

11.9. Continued fraction

```
op / = ([ ] real a, [ ] real b) real :
        ¢ the value of a / b is that of the continued fraction
        a1 / (b1 + a2 / (b2 + ... an / bn) ... ) ¢
    if lwb a = 1 ∧ lwb b = 1 ∧ upb a = upb b
    then (upb a = 0 | 0 | a [1] / (b [1] + a [2 : ] / b [2 : ]))
    fi
```

A **formula** using /:
*x1 / y1*

{The use of recursion may often be elegant rather than efficient as in the recursive procedure 11.8 and the recursive operation 11.9. See, however, 11.10 and 11.13 for examples in which recursion is of the essence.}

11.10. Formula manipulation

```
begin
    mode form = union (ref const, ref var, ref triple, ref call);
    mode const = struct (real value);
    mode var = struct (string name, real value);
    mode triple = struct (form left operand, int operator,
        form right operand);
    mode function = struct (ref var bound var, form body);
    mode call = struct (ref function function name, form parameter);
    int plus = 1, minus = 2, times = 3, by = 4, to = 5;
    heap const zero, one; value of zero := 0; value of one := 1;
    op = = (form a, ref const b) bool : (a | (ref const ec): ec :=: b | false);
    op + = (form a, b) form :
        (a = zero | b |: b = zero | a | heap triple := (a, plus, b));
    op − = (form a, b) form : (b = zero | a | heap triple := (a, minus, b));
    op × = (form a, b) form : (a = zero ∨ b = zero | zero |: a = one | b |: b = one | a |
        heap triple := (a, times, b));
    op / = (form a, b) form : (a = zero ∧ ¬ (b = zero) | zero |: b = one | a |
        heap triple := (a, by, b));
    op ↑ = (form a, ref const b) form :
        (a = one ∨ (b :=: zero) | one |: b :=: one | a | heap triple := (a, to, b));
    proc derivative of = (form e, ¢ with respect to ¢ ref var x) form :
        case e in
            (ref const): zero ,
            (ref var ev): (ev :=: x | one | zero) ,
            (ref triple et):
                case form u = left operand of et, v = right operand of et;
                    form udash = derivative of (u, ¢ with respect to ¢ x),
                        vdash = derivative of (v, ¢ with respect to ¢ x);
```

```
                operator of et
            in
                udash + vdash,
                udash - vdash,
                u × vdash + udash × v,
                (udash - et × vdash) / v,
                (v | (ref const ec): v × u ↑ (heap const c;
                    value of c := value of ec - 1; c) × udash)
            esac ,
        (ref call ef):
            begin ref function f = function name of ef;
                form g = parameter of ef; ref var y = bound var of f;
                heap function fdash := (y, derivative of (body of f, y));
                (heap call := (fdash, g)) × derivative of (g, x)
            end
    esac;
proc value of = (form e) real :
    case e in
        (ref const ec): value of ec ,
        (ref var ev): value of ev ,
        (ref triple et):
            case real u = value of (left operand of et),
                v = value of (right operand of et);
                operator of et
            in u + v, u - v, u × v, u / v, exp (v × ln (u))
            esac ,
        (ref call ef):
            begin ref function f = function name of ef;
                value of bound var of f := value of (parameter of ef);
                value of (body of f)
            end
    esac;
heap form f, g;
heap var a := ("a", skip), b := ("b", skip), x := ("x", skip);
¢ start here ¢
read ((value of a, value of b, value of x));
f := a + x / (b + x);
g := (f + one) / (f - one);
print ((value of a, value of b, value of x,
    value of (derivative of (g, ¢ with respect to ¢ x))))
end ¢ example of formula manipulation ¢
```

## 11.11. Information retrieval

```
begin
    mode ra = ref auth, rb = ref book;
```

```
mode auth = struct (string name, ra next, rb book),
 book = struct (string title, rb next);
ra auth, first auth := nil, last auth;
rb book; string name, title; int i; file input, output;
open (input, "", remote in); open (output, "", remote out);
putf (output, ($p
    "to enter a new author, type ""author"", a space,"x
    "and his name."l
    "to enter a new book, type ""book"", a space,"x
    "the name of the author, a new line, and the title."l
    "for a listing of the books by an author, type ""list"","x
    "a space, and his name."l
    "to find the author of a book, type ""find"", a new line,"x
    "and the title."l
    "to end, type ""end""""al$, "."));

proc update = void :
    if ra (first auth) :=: nil
    then auth := first auth := last auth := heap auth := (name, nil, nil)
    else auth := first auth;
        while ra (auth) :≠: nil
        do
            (name = name of auth | go to known | auth := next of auth)
        od;
        last auth := next of last auth := auth :=
            heap auth := (name, nil, nil);
known: skip
    fi;

do
try again:
    getf (input, ($c("author","book","list","find","end",""), x30al,
        80al$, i));

    case i in

        ¢ author ¢
        (getf (input, name); update),

        ¢ book ¢
        begin getf (input, (name, title)); update;
            if rb (book of auth) :=: nil
            then book of auth := heap book := (title, nil)
            else book := book of auth;
                while rb (next of book) :≠: nil
                do
                    (title = title of book
                    | go to try again | book := next of book)
                od;
```

```
                (title ≠ title of book
                | next of book := heap book := (title, nil))
            fi
        end,

        ¢ list ¢
        begin getf (input, name); update;
            putf (output, ($p"author:  "30all$, name));
            if rb (book := book of auth) :=: nil
            then put (output, ("no  publications", newline))
            else on page end (output,
                (ref file f) bool :
                    (putf (f, ($p"author:  "30a41k"continued"ll$, name));
                    true));
                while rb (book) :≠: nil
                do putf (output, ($l80a$, title of book)); book := next of book
                od;
                on page end (output, (ref file f) bool : false)
            fi
        end,

        ¢ find ¢
        begin getf (input, (loc string, title)); auth := first auth;
            while ra (auth) :≠: nil
            do book := book of auth;
                while rb (book) :≠: nil
                do
                    if title = title of book
                    then putf (output, ($l"author:  "30a$, name of auth));
                        go to try again
                    else book := next of book
                    fi
                od;
                auth := next of auth
            od;
            put (output, (newline, "unknown", newline))
        end,

        ¢ end ¢
        (put (output, (new page, "signed  off", close)); close (input);
            goto stop),

        ¢ error ¢
        (put (output, (newline, "mistake,  try  again")); newline (input))
    esac
  od
end
```

## 11.12. Cooperating sequential processes

```
begin int nmb magazine slots, nmb producers, nmb consumers;
    read ((nmb magazine slots, nmb producers, nmb consumers));
    [1 : nmb producers] file infile, [1 : nmb consumers] file outfile;
    for i to nmb producers do open (infile [i], "", inchannel [i]) od;
        ¢ 'inchannel' and 'outchannel' are defined in a surrounding
        range ¢
    for i to nmb consumers
    do open (outfile [i], "", outchannel [i]) od;
    mode page = [1 : 60, 1 : 132] char;
    [1 : nmb magazine slots] ref page magazine;
    int ¢ pointers of a cyclic magazine ¢ index := 1, exdex := 1;
    sema full slots = level 0, free slots = level nmb magazine slots,
        in buffer busy = level 1, out buffer busy = level 1;
    proc par call = (proc (int) void p, int n) void :
        ¢ call 'n' incarnations of 'p' in parallel ¢
        (n > 0 | par (p (n), par call (p, n - 1)));
    proc producer = (int i) void :
        do heap page page;
            get (infile [i], page);
            down free slots; down in buffer busy;
                magazine [index] := page;
                index modab nmb magazine slots plusab 1;
            up full slots; up in buffer busy
        od;
    proc consumer = (int i) void :
        do page page;
            down full slots; down out buffer busy;
                page := magazine [exdex];
                exdex modab nmb magazine slots plusab 1;
            up free slots; up out buffer busy;
            put (outfile [i], page)
        od;
    par (par call (producer, nmb producers),
        par call (consumer, nmb consumers))
end
```

## 11.13. Towers of Hanoi

```
for k to 8
do file f := stand out;
    proc p = (int me, de, ma) void :
        if ma > 0 then
            p (me, 6 - me - de, ma - 1);
            putf (f, (me, de, ma));
```

```
            ¢ move from peg 'me' to peg 'de' piece 'ma' ¢
            p (6 - me - de, de, ma - 1)
        fi;
    putf (f, ($l"k = "dl, n((2↑k+15)÷16)(2(2(4(3(d)x)x)x)l)$, k));
    p (1, 2, k)
od
```

## 12. Glossaries

### 12.1. Technical terms

Given below are the locations of the defining occurrences of a number of words which, in this Report, have a specific technical meaning. A word appearing in different grammatical forms is given once, usually as the infinitive. Terms which are used only within pragmatic remarks are enclosed within braces.

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

12.2. Paranotions

Given below are short paranotions representing the notions defined in this Report, with references to their hyper-rules.

**after-specification** 10.3.4.10.1.c
**alignment** 10.3.4.1.1.e
**alignment-code** 10.3.4.1.1.f
**alternate-CHOICE-clause** 3.4.1.d
**assignation** 5.2.1.1.a
**bits-denotation** 8.2.1.l
**bits-pattern** 10.3.4.7.1.a
**boolean-choice-pattern** 10.3.4.8.1.b
**boolean-marker** 10.3.4.4.1.b
**boolean-pattern** 10.3.4.4.1.a
**boundscript** 5.3.2.1.j
**call** 5.4.3.1.a
**case-clause** 3.4.1.p
**case-part-of-CHOICE** 3.4.1.i
**cast** 5.5.1.1.a
**character-glyph** 8.1.4.1.c
**character-marker** 10.3.4.6.1.b
**choice-clause** 3.4.1.n
**chooser-CHOICE-clause** 3.4.1.b
**closed-clause** 3.1.1.a
**coercee** 6.1.1.g
**coercend** 6.1.1.h

## 12.3. Predicates

Given below are abbreviated forms of the predicates defined in this Report.



## 12.4. Index to the standard prelude

## 12.5. Alphabetic listing of metaproduction rules

**ABC{942L} :: 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.**
**ADIC{542C} :: DYADIC ; MONADIC.**
**ALPHA{13B} :: 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.**
**BECOMESETY{942J} :: cum becomes ; cum assigns to ; EMPTY.**
**BITS{65A} :: structured with**
**row of boolean field SITHETY letter aleph mode.**
**BYTES{65B} :: structured with**
**row of character field SITHETY letter aleph mode.**
**CASE{34B} :: choice using integral ; choice using UNITED.**

**CHOICE{34A} :: choice using boolean ; CASE.**
**COLLECTION{A341C} :: union of PICTURE COLLITEM mode.**
**COLLITEM{A341D} :: structured with INSERTION field letter i digit one**
**procedure yielding integral field letter r letter e letter p**
**integral field letter p**
**INSERTION field letter i digit two mode.**
**COMARK{A341N} :: zero ; digit ; character.**
**COMMON{41A} :: mode ; priority ; MODINE identity ;**
**reference to MODINE variable ; MODINE operation ;**
**PARAMETER ; MODE FIELDS.**
**COMORF{61G} :: NEST assignation ; NEST identity relation ;**
**NEST LEAP generator ; NEST cast ; NEST denoter ;**
**NEST format text.**
**CPATTERN{A341I} :: structured with INSERTION field letter i**
**integral field letter t letter y letter p letter e**
**row of INSERTION field letter c mode.**
**DEC{123E} :: MODE TAG ; priority PRIO TAD ; MOID TALLY TAB ;**
**DUO TAD ; MONO TAM.**
**DECS{123D} :: DEC ; DECS DEC.**
**DECSETY{123C} :: DECS ; EMPTY.**
**DEFIED{48B} :: defining ; applied.**
**DIGIT{942C} :: digit zero ; digit one ; digit two ; digit three ; digit four ;**
**digit five ; digit six ; digit seven ; digit eight ; digit nine.**
**DOP{942M*} :: DYAD ; DYAD cum NOMAD.**
**DUO{123H} :: procedure with PARAMETER1 PARAMETER2 yielding MOID.**
**DYAD{942G} :: MONAD ; NOMAD.**
**DYADIC{542A} :: priority PRIO.**
**EMPTY{12G} :: .**
**ENCLOSED{122A} :: closed ; collateral ; parallel ; CHOICE ; loop.**
**EXTERNAL{A1A} :: standard ; library ; system ; particular.**
**FIELD{12J} :: MODE field TAG.**
**FIELDS{12I} :: FIELD ; FIELDS FIELD.**
**FIRM{61B} :: MEEK ; united to.**
**FIVMAT{A341L} :: mui definition of structured with row of**
**structured with integral field letter c letter p**
**integral field letter c letter o letter u letter n letter t**
**integral field letter b letter p row of union of structured**
**with union of PATTERN CPATTERN**
**structured with INSERTION field letter i**
**procedure yielding mui application field**
**letter p letter f mode GPATTERN void mode field letter p**
**INSERTION field letter i mode COLLITEM mode field**
**letter c mode field letter aleph mode.**
**FLEXETY{12K} :: flexible ; EMPTY.**
**FORM{61E} :: MORF ; COMORF.**
**FORMAT{A341A} :: structured with row of PIECE field letter aleph mode.**

**FPATTERN**{A341J} :: structured with INSERTION field letter i
procedure yielding FIVMAT field letter p letter f mode.
**FRAME**{A341H} :: structured with INSERTION field letter i
procedure yielding integral field letter r letter e letter p
boolean field letter s letter u letter p letter p
character field letter m letter a letter r letter k
letter e letter r mode.
**FROBYT**{35A} :: from ; by ; to.

**GPATTERN**{A341K} :: structured with INSERTION field letter i
row of procedure yielding integral field
letter s letter p letter e letter c mode.

**HEAD**{73B} :: PLAIN ; PREF ; structured with ; FLEXETY ROWS of ;
procedure with ; union of ; void.

**INDICATOR**{48A} :: identifier ; mode indication ; operator.
**INSERTION**{A341E} :: row of structured with
procedure yielding integral field letter r letter e letter p
union of row of character character mode field
letter s letter a mode.
**INTREAL**{12C} :: SIZETY integral ; SIZETY real.

**LAB**{123K} :: label TAG.
**LABS**{123J} :: LAB ; LABS LAB.
**LABSETY**{123I} :: LABS ; EMPTY.
**LAYER**{123B} :: new DECSETY LABSETY.
**LEAP**{44B} :: local ; heap ; primal.
**LENGTH**{65D} :: letter l letter o letter n letter g.
**LENGTHETY**{65F} :: LENGTH LENGTHETY ; EMPTY.
**LETTER**{942B} :: letter ABC ; letter aleph ; style TALLY letter ABC.
**LONGSETY**{12E} :: long LONGSETY ; EMPTY.

**MARK**{A341M} :: sign ; point ; exponent ; complex ; boolean.
**MEEK**{61C} :: unchanged from ; dereferenced to ; deprocedured to.
**MODE**{12A} :: PLAIN ; STOWED ; REF to MODE ; PROCEDURE ;
UNITED ; MU definition of MODE ; MU application.
**MODINE**{44A} :: MODE ; routine.
**MOID**{12R} :: MODE ; void.
**MOIDS**{46C} :: MOID ; MOIDS MOID.
**MOIDSETY**{47C} :: MOIDS ; EMPTY.
**MONADIC**{542B} :: priority iii iii iii i.
**MONAD**{942H} :: or ; and ; ampersand ; differs from ; is at most ;
is at least ; over ; percent ; window ; floor ; ceiling ;
plus i times ; not ; tilde ; down ; up ; plus ; minus ;
style TALLY monad.
**MONO**{123G} :: procedure with PARAMETER yielding MOID.
**MOOD**{12U} ::
PLAIN ; STOWED ; reference to MODE ; PROCEDURE ; void.
**MOODS**{12T} :: MOOD ; MOODS MOOD.

**MOODSETY{47B} :: MOODS ; EMPTY.**
**MORF{61F} :: NEST selection ; NEST slice ; NEST routine text ;**
**NEST ADIC formula ; NEST call ;**
**NEST applied identifier with TAG.**
**MU{12V} :: muTALLY.**
**NEST{123A} :: LAYER ; NEST LAYER.**
**NOMAD{942I} :: is less than ; is greater than ; divided by ; equals ;**
**times ; asterisk.**
**NONPREF{71B} :: PLAIN ; STOWED ;**
**procedure with PARAMETERS yielding MOID ; UNITED ; void.**
**NONPROC{67A} :: PLAIN ; STOWED ; REF to NONPROC ;**
**procedure with PARAMETERS yielding MOID ; UNITED.**
**NONSTOWED{47A} :: PLAIN ; REF to MODE ; PROCEDURE ; UNITED ;**
**void.**
**NOTETY{13C} :: NOTION ; EMPTY.**
**NOTION{13A} :: ALPHA ; NOTION ALPHA.**
**NUMERAL{810B*} :: fixed point numeral ; variable point numeral ;**
**floating point numeral.**
**PACK{31B} :: STYLE pack.**
**PARAMETER{12Q} :: MODE parameter.**
**PARAMETERS{12P} :: PARAMETER ; PARAMETERS PARAMETER.**
**PARAMETY{12O} :: with PARAMETERS ; EMPTY.**
**PART{73E} :: FIELD ; PARAMETER.**
**PARTS{73D} :: PART ; PARTS PART.**
**PATTERN{A341G} :: structured with**
**integral field letter t letter y letter p letter e**
**row of FRAME field**
**letter f letter r letter a letter m letter e letter s mode.**
**PICTURE{A341F} :: structured with union of PATTERN CPATTERN**
**FPATTERN GPATTERN void mode field letter p**
**INSERTION field letter i mode.**
**PIECE{A341B} :: structured with integral field letter c letter p**
**integral field letter c letter o letter u letter n letter t**
**integral field letter b letter p**
**row of COLLECTION field letter c mode.**
**PLAIN{12B} :: INTREAL ; boolean ; character.**
**PRAGMENT{92A} :: pragmat ; comment.**
**PRAM{45A} :: DUO ; MONO.**
**PREF{71A} :: procedure yielding ; REF to.**
**PREFSETY{71C*} :: PREF PREFSETY ; EMPTY.**
**PRIMARY{5D} :: slice coercee ; call coercee ; cast coercee ;**
**denoter coercee ; format text coercee ;**
**applied identifier with TAG coercee ; ENCLOSED clause.**
**PRIO{123F} :: i ; ii ; iii ; iii i ; iii ii ; iii iii ; iii iii i ; iii iii ii ; iii iii iii.**
**PROCEDURE{12N} :: procedure PARAMETY yielding MOID.**
**PROP{48E} :: DEC ; LAB ; FIELD.**

PROPS{48D} :: PROP ; PROPS PROP.
PROPSETY{48C} :: PROPS ; EMPTY.

QUALITY{48F} :: MODE ; MOID TALLY ; DYADIC ; label ; MODE field.

RADIX{82A} :: radix two ; radix four ; radix eight ; radix sixteen.
REF{12M} :: reference ; transient reference.
REFETY{531A} :: REF to ; EMPTY.
REFLEXETY{531B} :: REF to ; REF to flexible ; EMPTY.
ROWS{12L} :: row ; ROWS row.
ROWSETY{532A} :: ROWS ; EMPTY.

SAFE{73A} :: safe ; MU has MODE SAFE ; yin SAFE ; yang SAFE ;
    remember MOID1 MOID2 SAFE.
SECONDARY{5C} ::
    LEAP generator coercee ; selection coercee ; PRIMARY.
SHORTH{65E} :: letter s letter h letter o letter r letter t.
SHORTHETY{65G} :: SHORTH SHORTHETY ; EMPTY.
SHORTSETY{12F} :: short SHORTSETY ; EMPTY.
SITHETY{65C} :: LENGTH LENGTHETY ; SHORTH SHORTHETY ; EMPTY.
SIZE{810A} :: long ; short.
SIZETY{12D} :: long LONGSETY ; short SHORTSETY ; EMPTY.
SOFT{61D} :: unchanged from ; softly deprocedured to.
SOID{31A} :: SORT MOID.
SOME{122B} :: SORT MOID NEST.
SORT{122C} :: strong ; firm ; meek ; weak ; soft.
STANDARD{942E} :: integral ; real ; boolean ; character ; format ; void ;
    complex ; bits ; bytes ; string ; sema ; file ; channel.
STOP{A1B} :: label letter s letter t letter o letter p.
STOWED{12H} :: structured with FIELDS mode ;
    FLEXETY ROWS of MODE.
STRONG{61A} :: FIRM ; widened to ; rowed to ; voided to.
STYLE{133A} :: brief ; bold ; style TALLY.

TAB{942D} :: bold TAG ; SIZETY STANDARD.
TAD{942F} :: bold TAG ; DYAD BECOMESETY ;
    DYAD cum NOMAD BECOMESETY.
TAG{942A} :: LETTER ; TAG LETTER ; TAG DIGIT.
TAILETY{73C} :: MOID ; FIELDS mode ; PARAMETERS yielding MOID ;
    MOODS mode ; EMPTY.
TALLETY{542D} :: TALLY ; EMPTY.
TALLY{12W} :: i ; TALLY i.
TAM{942K} :: bold TAG ; MONAD BECOMESETY ;
    MONAD cum NOMAD BECOMESETY.
TAO{45B} :: TAD ; TAM.
TAX{48G} :: TAG ; TAB ; TAD ; TAM.
TERTIARY{5B} :: ADIC formula coercee ; nihil ; SECONDARY.
THING{13D} :: NOTION ; (NOTETY1) NOTETY2 ;
    THING (NOTETY1) NOTETY2.

**TYPE**{A341P} :: **integral ; real ; boolean ; complex ; string ; bits ;**
**integral choice ; boolean choice ; format ; general.**
**UNIT**{5A} :: **assignation coercee ; identity relation coercee ;**
**routine text coercee ; jump ; skip ; TERTIARY.**
**UNITED**{12S} :: **union of MOODS mode.**
**UNSUPPRESSETY**{A341O} :: **unsuppressible ; EMPTY.**
**VICTAL**{46A} :: **VIRACT ; formal.**
**VIRACT**{46B} :: **virtual ; actual.**
**WHETHER**{13E} :: **where ; unless.**