Limited Circulation
(members of WG 2.1 only)

Another Proposal for ALGOL 67 ×

by

G.Goos, H.Scheidig,
G.Seegmüller, H.Walther

Rechenzentrum der Technischen Hochschule München
Leibniz-Rechenzentrum der Bayerischen Akademie
der Wissenschaften München

Report 6704

× in the spirit of the approach of A.van Wijngaarden using earlier documents of him and other members of the Working Group 2.1 of IFIP

Contents

o.	Introduction	1
o.1.	Aims and principles of design	1
o.2.	Differences with regard to ALGOL 6o	3
o.2.1.	Primitive data types	3
o.2.2.	Connected data types	5
o.2.3.	References	6
o.2.4.	The free feature	6
o.2.5.	Carriers	7
o.2.6.	Declarations	7
o.2.7.	Procedures and parameters	7
o.2.8.	Expressions	8
o.2.9.	Case expressions	9
o.2.1o.	Iterative expressions	9
o.2.11.	Environment enquiries	9
o.2.12.	Input / Output procedures	9
o.2.13.	Parallelism	1o
1.	Forms of the language and introduction to the definition method	11
1.1.	Forms of the language	11
1.1.1.	The strict language	11
1.1.2.	The extended language	12
1.1.3.	The representation language	12
1.2.	Introduction to the definition method	13
1.2.1.	The production rules of the metalanguage	13
1.2.2.	The production forms	14
1.2.3.	The production rules of the strict language	14
2.	Metalanguage and the construction of productions	16
2.1.	Syntax	16
2.2.	The construction of the production rules of the strict language	18

3.	Basic concepts of the strict language	2o
3.1.	Language, program and computer	2o
3.2.	Objects	21
3.3.	Quantities	21
3.4.	Entities	22
3.5.	Denotations	22
3.6.	Attributes of values	22
3.7.	correspondences of values	25
3.8.	Elementary actions	26
3.9.	Operations	26
3.1o.	The computation	27
3.11.	Implementations	28
4.	Basic constituents of the language	29
4.1.	Basic symbols	29
4.2.	letters	31
4.3.	Value denotation symbols	31
4.4.	Operator symbols	32
4.5.	Declaration symbols	33
4.6.	Syntactical symbols	34
4.7.	Sequencing symbols	34
4.8.	Extra symbols	34
4.9.	Identifiers	35
4.1o.	Numbers	35
5.	Program and phrases	37
5.1.	Program	37
5.2.	Phrases	37
5.2.2.1.	Interruption of the elaboration of a phrase	38
5.2.2.2.	Termination of the elaboration of a phrase	39
5.2.2.3.	Completion of the elaboration of a phrase	39
5.3.	Unitary blocks	4o
5.3.2.1.	Initiation of unitary blocks	41

5.3.2.2.	Completion of the elaboration of a unitary block and its value	41
5.4.	Parallel blocks	43
5.5.	Conditional expressions	43
5.6.	Jumps	45
5.7.	Determination of successors	45
5.8.	Identification of entities	46
6.	Declarations	48
6.1.	Declarators	49
6.1.3.	Defining occurrences	51
6.2.	Record type declarations	51
6.3.	Carrier declarations	52
6.3.3.	Descriptors for arrays	54
6.3.4.	Descriptors for records	55
6.4.	Procedure declarations	56
7.	Proper expressions	58
7.1.	Assignements	58
7.2.	Plain expressions	62
7.3.	Simple arithmetic expressions	63
7.4.	Simple boolean expressions	65
7.5.	Simple binal expressions	68
7.6.	Simple alphameric expressions	69
7.7.	Primaries	71
7.8.	Direct denotations	71
7.9.	Value denotations	74
7.10.	Record generators	77
7.11.	Named primaries	78
7.12.	Procedure designators	79
7.13.	Direct primaries	83
7.14.	Field selections	83
7.15.	Evaluated primaries	84

7.16.	Indexed primaries	85
7.17.	Lengthened primaries	88
7.18.	Shortened primaries	89
7.19.	Narrowed primaries	89
7.2o.	Widened primaries	9o
7.21.	Case expressions	91
8.	Representations	93
9.	Further extensions	98
1o.	Environment inquiries	1oo
11.	Input/output	1oo
12.	Examples	1o1

DRAFT

The Algorithmic Language ALGOL 67

0. Introduction

0.1. Aims and principles of design

In defining the Algorithmic Language ALGOL 67 the members of the Working Group 2.1. of IFIP express their belief in the value of a common programming language serving many people in many countries. The language is envisaged for executing algorithms on a variety of different computers, for communicating algorithms, and for use in teaching algorithms to students.

The members of the group, influenced by the experience of several years of working with ALGOL 60 and other programming languages, were guided mainly by the following principles of design for the new language:

a) Completeness and clarity of description. By way of example, WG 2.1 wants to contribute to the solution of the problem of defining a language as clearly and as completely as possible. It is recognized, however, that the method adopted may be inconvenient for the uninitiated reader.

b) Conceptual economy. The number of concepts was kept as low as possible. In this way the language becomes much easier to describe, to learn and to implement.

c) Power by generality. The expressive power of the language was achieved by the generality of the concepts introduced. None of the generalities, however, seems to be unnecessary.

d) Appropriateness of the features. It has been kept in mind that there exists the necessity of balancing the language features, on the one hand, with the facilities available on present day computers on the other hand. The ideal was that for any given algorithmic step which is within the power of the language, there corresponds exactly one feature which lends itself naturally for that purpose, and that this instance of using the feature can be implemented efficiently.

Among others there are several principles of a more technical nature which may be regarded as being consequences of the general lines of design listed above and which were adopted by the group:

e) Static type checking. Apart from the free feature, (cf. o.2.4), the use of which is restricted to cases where it is indispensable, data types and the operations on them are chosen so that in an implementation which distinguishes between compile phase and object phase the object program need never do branching on types, whereas checking of types is only necessary in certain cases of procedure entry.

f) Possibility of independent compilation. The procedure mechanism allows appropriate specific handling of the actual and formal parameters in the compile phase, even if an independently compiled procedure is called.

g) Ease of loop optimization. The way in which iterative processes can be formulated in the language is suited for straightforward application of well-known optimization techniques.

h) Static control over dynamic transfers of control. The language elements commonly known as labels are non-manipulable things in this language, i.e. the only operator applicable to them is the goto-operator. This allows a considerable reduction of administrative work at object time.

Language features which are novel or underwent changes with respect to ALGOL 6o are introduced to the reader in the following section.

o.2. Differences with regard to ALGOL 6o

Before giving a summary of the new features it may be noted that most of the powerful concepts and notions of ALGOL 6o have entered into this language, although they often appear in a more general form. This is especially true for the notions of block, declaration, and expression. The latter now includes the concept of a statement of ALGOL 6o.

On the other hand, there are a few features of ALGOL 6o which have been eliminated, at least in the old form. Among them are the own concept, the possibility of writing designational expressions and integer labels.

o.2.1. Primitive data types

There are six specific primitive data types in the language, namely

integral:	for integral numbers lying within a certain range,
real:	for rational numbers lying within a certain range,
complex:	for complex rational numbers lying within a certain range,
binal:	for sequences of binal values each being either o or 1,
alphameric:	for sequences of alphameric data each being a character,
boolean:	for the boolean values true and false.

The capacity of each primitive type (except in the trivial case for boolean) is not specified. It is assumed that it will differ from implementation to implementation and that a primitive datum will normally be represented by a small number of computer words or a fraction of a computer word. The absence

o.2.1. continued 1

of the possibility to specify the capacity of the primitive types is the result of careful considerations balancing efficiency vs. exchangeability of programs. It is recognized that there is a fundamental difference with respect to capacity between the arithmetic types, i.e. integral, real, complex, and those intended primarily for non-numerical work, i.e. binal and alphameric, but it was felt that it is wiser to have a programmer ask for the capacity of the primitive types of the implementation he wants to work with, rather than imposing a specification which may result in inefficient code. Moreover, by means of properly posed environment enquiries (cf. o.2.11.) it is possible to write programs running efficiently on different computers.

The language provides for increasing the capacity of the primitive types (except boolean) by means of the long symbol. When declaring a quantity, this symbol may be added in an iterated manner and the capacity of a quantity having one more long symbol in its declaration than another one is assumed at least not to be less than the capacity of the latter. The implementor has to specify the capacities corresponding to the number of long symbols in front of the primitive type symbols.

The objects corresponding to the above six primitive data types are assumed to have an existence independent of the program itself. The meaning of their representations is independent of the context within which they are processed (e.g. -3.14, b 1o111, "a2q34"). It is a considerable difference from ALGOL 6o that values of type alphameric (the strings of ALGOL 6o) are sequences of characters rather than sequences of basic symbols of the language.

Automatic type conversions take place from

integral to real,
integral to complex,
real to complex,

and in certain cases from a specific type to a longer variant of the same type. For all other conversions the explicit use of operators or standard procedures is obligatory. In this way an important source of possible inefficiency is clearly indicated to the programmer.

0.2.2. Connected data types

In the language there are two ways of grouping data to form higher units of information. The first one is well-known from ALGOL 60. Arrays may be formed essentially in the old way. In addition there exists the feature of "trimming" an array which allows convenient manipulation of linear subarrays of a given array. Moreover, there is an assignment expression for arrays and an array value denotation.

From the programmer's point of view, arrays are suited for grouping rectangularly indexed data of the same primitive type and of fixed number of elements. To meet the needs of a wider class of applications, a much more flexible way of handling data groups was introduced into the language by means of the record concept. A record may be viewed as being a finite set of

primitive data,
arrays,
other records,
and references (cf. 0.2.3.)

in some prescribable order. Thus the configuration of a record is in itself a new type. This type is declared by a record type declaration. Variables or constants of this type may be declared by record declarations. By these record declarations, in contrast to all other declarations of the language, the number of objects of the corresponding record type is not fixed. By means of the so-called record generator new instances of the same record type may be dynamically generated at will. A single constituent of a record, a field, may be accessed by applying a so-called field selector.

o.2.2. continued

A record is an object, the lifetime of which is not identical with the lifetime of one of the blocks which contain the corresponding record type declaration or the record declaration. It is generated after these blocks come to existence and it may be lost before the blocks are left, e.g. upon assignment of a new record to a record variable. Therefore a substantial extension of the implementation techniques in comparison with ALGOL 6o is necessary in order to cope with the whole language in an efficient manner.

o.2.3. References

In handling quantities inside a computer, it is a common practice in data processing either to work with the quantity itself (e.g. to transport it) or to manipulate a pointer to it. Depending on a variety of circumstances, the one or the other of the two methods may be far more efficient. This is reflected in the language by placing both techniques at the programmer's disposal. Thus references are allowed to quantities of primitive and of connected type and also to procedures (cf. o.2.7.). References may be referenced again. In this way a hierarchy of "levels" of references is obtained where level zero indicates quantities which are not references.

There is a fundamental difference between references and the other objects mentioned so far. Whereas the latter possess representations which are independent from the context within which they appear, this is not true for references. Therefore assignments of references have certain natural restrictions depending on the context within which they appear.

o.2.4. The free feature

In order to allow the formulation of parts of algorithms which deal with quantities, the types of which may vary from execution to execution of that part of the program, free re-

ferences were introduced. A reference to a quantity of any type but of fixed level may be assigned to a free reference. Assignments in connection with free references, however, may only occur within a so-called conformity clause. This feature allows a static check to eliminate all dynamic type checking except the ones explicitly indicated in the program.

0.2.5. Carriers

In ALGOL 60, identifiers were used either to denote variables of certain types or constants of certain kinds. It was, however, not possible to denote e.g. a constant of type real. The declaration itself contained the one or the other alternative implicitly. Because of the importance of this information in the new language, the programmer has the choice of specifying an identifier to denote either a constant or a variable except in the case of labels and procedures. In the sequel, the notion of a"carrier" is used to mean variables as well as constants.

0.2.6. Declarations

All carriers must be declared. In the case of a variable an initialization may or ,in special cases,must take place, whereas in the case of a constant it is obligatory. Note that the procedure declaration of ALGOL 60 is a constant declaration in this terminology. Contrary to ALGOL 60, declarations of carriers are"elaborated" sequentially, where "elaboration" is a technical term meaning execution and evaluation as well.

0.2.7. Procedures and parameters

The specification of formal parameters in this language are simply declarations. Upon call of the procedure, they are converted to initialized declarations of constants by means of the actual parameters. The compatibility rules of ALGOL 60 reduce to the rule that everything is allowed at a specific

0.2.7. continued

actual parameter position which is allowed as initializing expression for the corresponding constant, i.e. the compatibility problem is (apart from the constant feature) equivalent to the assignment problem. Note that a label is not an assignable quantity. Jumping out of a procedure via a procedure parameter has to be performed by means of a procedure designator. A"procedure torso", however, is an assignable quantity, and therefore may appear as an actual parameter. This usage corresponds exactly to the use of name parameters in ALGOL 60.Moreover, the language provides not only for value parameters in the sense of ALGOL 60 but also for so-called reference parameters.

0.2.8. Expressions

The notion of an expression of ALGOL 60 has been extended. In particular, it includes certain classes of ALGOL 60 statements, e.g. blocks. The possibility of the so-called intermediate assignment is, among others, a natural consequence of this fact. In some cases, the elaboration of certain parts of an expression is said to take place in an "unspecified order". This is used to rule out side effects, e.g. of procedures.

Upon the completion of its elaboration, each expression possesses a value. Therefore blocks have always a value, if they are not left by means of a jump.

There is no possibility of jumping into expressions(especially not into those which correspond to certain classes of ALGOL 60 statements, e.g. conditional statements and blocks which are compound statements in the sense of ALGOL 60).

The meaning of boolean expressions was changed in the sense that the elaboration of an expression containing dyadic boolean operators does not necessarily involve the elaboration of all operands concerned.

0.2.9. Case expressions

A powerful generalization of the conditional statements and expressions of ALGOL 60 is the case construction which follows very closely well-known implementation techniques of multi-way forks.

0.2.10. Iterative expressions

The for statement of ALGOL 60 has been improved in several ways. The expressions appearing in the for clause (in the sense of ALGOL 60) are elaborated only once, namely upon entry into the for expression. Moreover, the for expression constitutes a block with the for counter being a local quantity of that block to which no assignments other than those induced by the for clause are possible.

0.2.11. Environment enquiries

Among the standard procedures of the language there are several procedures which allow the program to ask for certain implementation parameters such as the maximum number of characters in an alphameric carrier or the size of the so-called machine epsilon for reals, etc.

0.2.12. Input/output procedures

The set of input/output procedures consists of three subsets, first, procedures transforming a format string into another string which may be used as a control string for certain procedures, second, procedures which transform data from internal to external representations or vice versa, and finally, procedures performing the proper I/O transmission.

0.2.13. Parallelism

There is a special pair of parentheses for enclosing a sequence of blocks. They indicate that the blocks enclosed may be executed in parallel. This feature may be of use for implementations capable of doing I/O overlapping, multiprogramming and multiprocessing.

No precautions, however, are made to prevent the programmer from algorithmic errors resulting in a dynamic hang-up situation within blocks which permit parallel elaboration.

1. Forms of the language and introduction to the definition method

1.1. Forms of the language

The algorithmic language ALGOL 67 is a language in which programs (cf. 5.1.), which describe algorithms, can be written and which can be executed by means of a computer. It is defined in three stages called the "strict language", "extended language", and "representation language".

1.1.1. The strict language

Most of the following chapters are concerned with the definition of the strict language.

The syntax of the strict language is defined by means of a set P of "production rules". This set P has an infinite number of members. It is generated by two other sets each of which is finite. The first set consists of "production forms" from which production rules, members of P, are obtained by replacing so-called "metanotions" (cf. 1.2.) by sequences of letters. These sequences are generated from the second set which consists of production rules for the so-called "metalanguage" (cf. 1.2.).

The semantics of the strict language is described in the English language. The connection between syntax and semantics is obtained as follows: if words occur in sections called Semantics which appear also as notions, metanotions, or parts of notions in sections called Syntax, then these words denote either these notions or productions of these notions (cf. 1.2.). The structure of the English language may cause deviations in the appearance of a notion which occurs under Semantics. Therefore partially capitalized forms, plural forms, split forms, and generalizing forms must be properly interpreted. Under the heading Syntax several productions for notions have been included which are irrelevant from a syntactical point of view. These productions are preceded by an asterisk.

1.1.2. The extended language

The strict language is not very well suited for practical use by a human being, because, for clarity of definition it consists of sequences of notions (separated by commas), rather than sequences of representations of symbols which would suggest an appropriate intuitive meaning. Moreover for ease of definition, the strict language contains sequences of notions which are very uncommon from a traditional point of view.

It is the extended language which is free from the latter inconvenience. In order to define it, the following chapters frequently contain sections in which so-called "extensions" are given. Each extension is a rule for a mechanical replacement of one sequence of notions by another one. In this way the definition of the extended language proceeds as the definition of the strict language goes on. No new semantics of the extended language need be given, because the extended language differs from the strict language by notational changes, only. Certain features, e.g. iterative expressions (cf. 9.) are defined merely by extension.

1.1.3. The representation language

In order to remove the first inconvenience mentioned above so-called representations are given (cf. 8.) which consist of certain typographical marks and correspond in a unique manner to the terminal symbols (cf. 1.2.) of the strict language or to the symbols introduced by extensions. A program in the representation language may be obtained by replacing all notions occurring in a program of the strict or extended language by their representations and deleting all commas separating these notions.

Each version of the language in which representations are used which are sufficiently close to the given representations

1.1.3. continued

to be identified with them without further elucidation is equally well entitled to be called a representation language.

1.2. Introduction to the definition method

1.2.1. The production rules of the metalanguage

In section 2.1. a set of "rules" is given from which the production rules of the metalanguage may be obtained by means of the following process:

> Each rule containing a " / " is replaced by two new rules. The first new rule consists of the part of the rule before that " / ", followed by a point. The second new rule consists of the part of the rule up to and including the " ::= ", followed by the part of the rule after that " / ".
> This action is repeated until all " / " have been eliminated.

The rules obtained in this way are called the "production rules" of the metalanguage. They consist of sequences of capital and possibly small letters, one " ::= ",and have a point at their right end.

Each production rule of the metalanguage begins with a sequence of capital letters followed by a " ::= ". Such a sequence of capital letters is called a"metanotion"and the production rule is called a production rule for that metanotion.

A "metanotion list" is either a metanotion or a sequence of small letters, possibily separated by blanks, or consists of a metanotion list followed by a blank followed by either a metanotion or a sequence of small letters possibly separated by blanks, or it is empty.
If a metanotion list appears after the " ::= " in a production rule for a metanotion, then this list is called a "direct production" of that metanotion.

1.2.1. continued

A "production" of a metanotion is either a direct production of it or a metanotion list obtained by replacing in a production of that metanotion a constituent metanotion by a direct production of this constituent metanotion.

A "terminal production" of a metanotion is a production of that metanotion which contains no metanotion.

1.2.2. The production forms

In chapter 4-7 under the heading Syntax a set of rules is given from which the so-called production forms may be obtained. Together with the metalanguage the production forms serve to construct the production rules of the strict language (cf.2.2.).

The production forms are obtained in part from the above mentioned set of rules by performing the process described at the beginning of 1.2.1.

1.2.3. The production rules of the strict language

A sequence of small letters possibly separated by one or more blank spaces,or by a change to a new line or to a new page,is called a "notion", if within a given context it is the sequence with the maximum number of letters satisfying that condition.

A "notion list" is either a notion or is a notion list,followed by a comma,followed by a notion.

A production rule of the strict language is a production rule for a notion.It consists of that notion,followed by " ::= ", followed by a notion list other than that notion,followed by a point.This notion list is called a "direct production" of that notion.

A "production" of a notion is either a direct production of it,

1.2.3. continued

or a notion list which is obtained by replacing in a production of that notion a constituent notion by a direct production of it.

A "terminal symbol" is a notion for which no production rule is given.

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

2. Metalanguage and the construction of productions

2.1. Syntax

2.1.1. ACTAL ::= actual / formal.
2.1.2. ALPHAMERIC ::= LONG alphameric.
2.1.3. ARITHMETIC ::= NONCOMPLEX / COMPLEX.
2.1.4. ARRAY ::= array of / array of ARRAY.
2.1.5. BINAL ::= LONG binal.
2.1.6. CARRIER ::= MODE / constant SIMPLE.
2.1.7. COMPLEX ::= LONG complex.
2.1.8. CONNECTED ::= STRUCTURED / RECORD.
2.1.9. CONSTANT ::= constant ORDINARY.
2.1.1o. EMPTY ::= .
2.1.11. ENTITY ::= CARRIER / PROCEDURE / RECORD type / ORDINARY field / label.
2.1.12. FREE ::= free / constant free.
2.1.13. FULL ::= PRIMITIVE / CONNECTED / SPECIAL.
2.1.14. GRADED ::= reference to a / reference to a GRADED / reference to a constant GRADED.
2.1.15. INTEGRAL ::= LONG integral.
2.1.16. KIND ::= TYPE / SPECIAL.
2.1.17. LEVELED ::= GRADED FULL / GRADED constant PRIMITIVE / GRADED constant CONNECTED / GRADED constant free.
2.1.18. LONG ::= EMPTY / long LONG.
2.1.19. LOWPER ::= lower / upper.
2.1.2o. MODE ::= ORDINARY / constant CONNECTED.
2.1.21. NONBOOLEAN ::= ARITHMETIC / ALPHAMERIC / BINAL.
2.1.22. NONCOMPLEX ::= INTEGRAL / REAL.
2.1.23. NONCONSTANT ::= ORDINARY / free.
2.1.24. NONFREE ::= TIED / array of NONFREE / reference to a NONFREE / reference to a constant NONFREE / constant RECORD / constant array of NONFREE.

2.1. continued

2.1.25. NONINTEGRAL ::= REAL / COMPLEX.
2.1.26. NONPRIMITIVE ::= LEVELED / CONNECTED / constant CONNECTED / FREE.
2.1.27. ORDINARY ::= SIMPLE / CONNECTED.
2.1.28. PARAMETER ::= CARRIER parameter / PROCEDURE parameter.
2.1.29. PARAMETERS ::= PARAMETER / PARAMETERS and PARAMETER.
2.1.3o. PRIMITIVE ::= NONBOOLEAN / boolean.
2.1.31. PROCEDURE ::= procedure with a MODE result / procedure with a MODE result and PARAMETERS.
2.1.32. RAISED ::= GRADED / array of RAISED / GRADED array of RAISED.
2.1.33. REAL ::= LONG real.
2.1.34. RECORD ::= record with an ORDINARY field / RECORD and an ORDINARY field.
2.1.35. SIMPLE ::= PRIMITIVE / LEVELED.
2.1.36. SORT ::= MODE / FREE.
2.1.37. SPECIAL ::= free / procedure with a MODE result and an undetermined number of parameters.
2.1.38. STRUCTURED ::= array of ORDINARY.
2.1.39. TIED ::= PRIMITIVE / RECORD / procedure with a MODE result and an undetermined number of parameters.
2.1.4o. TOKEN ::= binary digit / octal digit / hexadecadic digit / digit / character.
2.1.41. TYPE ::= integral / real / complex / binal / alphameric / boolean.

2.2. The construction of the production rules of the strict language

Starting with the process explained in 1.2.1. and 1.2.2. the construction of the production rules of the strict language is described by the following "process of consistent substitution" :

> If a production form contains a metanotion, then this production form is replaced by a set of production forms in each of which the metanotion is consistently replaced by the same terminal production. This process is successively repeated for each member of the set obtained, until a set of production rules of the strict language results.

Since some metanotions have an infinite number of productions, the number of production rules of the strict language obtained by the process described is infinite.

2.3. Remarks

From a syntactical point of view the strict language consists of all terminal productions of the specific notion "program" (cf.5.1.). Note, that not all members of this set, however, satisfy the restrictions given in chapter 4-7 under the heading Semantics.

For reasons of abbreviation, words which are notions or metanotions frequently are used in places where productions of those notions or metanotions are meant.

2.4. Examples

Some productions of the metanotion NONBOOLEAN are:

2.4. continued

ARITHMETIC
NONCOMPLEX
REAL
LONG real
long EMPTY real
long real

The last production is a terminal production of NONBOOLEAN.

A production rule of the strict language derived from the production form 7.1.1.1. is

long real assignment :: long real destination,
becomes symbol,long real source.

whereas the sequence

long real assignment :: real destination,
becomes symbol,long real source.

is not a production rule of the strict language since the metanotion NONFREE was replaced at different places by different terminal productions of it.

3. Basic concepts of the strict language

In this chapter the method is explained which is used to describe the semantics of the strict language. Certain technical terms are introduced which occur later in sections called Semantics. There is also an explanation of the data types.

3.1. Language, program and computer

A specific notion of the strict language is "program". In chapters 4 - 7 under the heading Semantics, a so-called meaning is associated with each production of program. This is done by means of sentences in the English language, which describe "actions" taken by a hypothetical "computer" which comprises a finite but extensible set of "items". Many of these items are "objects" between which, at any given time, certain "relationships" may hold. Establishing or changing such relationships, creating, deleting or changing objects are some of the possible actions the computer may take. Each action, however, may be replaced by any other process which causes the same effect.

Each action taken by the computer is either "elementary" or not. If it is not elementary, then it consists of two or more actions which take place either "in series", or "in unspecified order", or "in parallel".

A program is a certain "expression". An expression consists of "phrases". A phrase is either a "declaration" or itself an expression. The action taken by the computer according to a specific phrase is called the "elaboration" of that phrase. Correspondingly, elaborations may take place either "in series", or "in unspecified order", or "in parallel". The beginning of the elaboration of a phrase is called the "initiation" of the elaboration of that phrase.

"Elaboration in series" means actions taking place one after the other. "Elaboration in unspecified order" means an unspecified merging in time of all actions concerned. "Elaboration in parallel"

means initiating the elaboration of the corresponding phrases at the same time. After this initiation all actions involved take place in unspecified order.

3.2. Objects

Objects are either "values", expressions of certain classes, or "procedure torsos". They are distinguished from other items the computer may comprise, by the property, that they can be manipulated by the actions of the computer when elaborating a program.

Some values, called "primitive values", have meaning independent of the "computation", i.e. the elaboration of the program. The other values, called "appellations", derive their meaning from the computation. Appellations are also called "references" or "names".

3.3. Quantities

Each appellation is paired with an object. Such a pair is called a "quantity". The appellation is said "to refer" to that object with which it forms the quantity. The appellation and the object are also said "to belong" to that quantity. On the other hand a quantity is said "to possess" an appellation and an object.

A quantity is either a "carrier", or a "procedure". A carrier is either a "constant" or a "variable". If it is a variable, then the object belonging to the quantity may be replaced by another object. This action is known as "assignment". This possibility does not exist for a constant or a procedure.

The object belonging to a carrier is also called the "value of that carrier". In the case of a procedure it is called the procedure torso.

3.4. Entities

An "entity" is either a quantity, or a "label", or a "record type", or a "field selector".

3.5. Denotations

"Denotations" are either "primitive value denotations" or "identifiers".

A primitive value denotation denotes a primitive value. This relationship holds permanently, independently of the computation.

An identifier may denote an entity. This relationship is established by a "definition", i.e. a declaration or a "label definition".

3.6. Attributes of values

A value is of some "mode" and has a "size". This size is the intrinsic property that distinguishes the value from all other values of that mode.

There exists one value, the "neutral value", which is of any mode. Its size differs from the size of all other values.

Each other value is of a specific mode characterized by the following attributes: "type", possibly "length", "level", "structure" and possibly "constancy". A value is called a primitive value, if it is of primitive type and has structure and level zero.

A type is either a "primitive type" or a record type. A record type is an ordered set of modes. There exist six classes of primitive values. These classes may be characterized as follows:

integral:	a finite set of integers,
real:	a finite set of rational numbers,
complex:	a finite set of complex rational numbers,
binal:	a finite set of finite sequences of binary digits,
alphameric:	a finite set of finite sequences of characters,
boolean:	the values "true" and "false".

3.6. continued 1

The types integral, real and complex are known as "arithmetic types".

The types integral, real, complex, binal and alphameric are also called "nonboolean types". Values of these types may be kept in the computer with different "length". With each nonboolean type there is always associated a certain length. The length is a nondecreasing integral function of the number of long symbols appearing, e.g. in "direct denotations" and in declarators for that nonboolean type. To each nonboolean type with a given length belongs a finite set of primitive values with distinct sizes. The "reach" of a nonboolean type of a certain length is given by a collection of several primitive values of the same type, characterizing the mentioned set by means of their sizes. The reach for the type real with a certain length, e.g., could be given by three real numbers indicating the maximum absolute value, the minimum absolute value and the greatest number that, if being added to one, yields one. For the type alphameric with a certain length, on the other hand, an arbitrary string with maximum number of characters for that length will suffice.

The sets of primitive values belonging to the different lengths of a nonboolean type form a sequence of subsets such, that the set belonging to a certain length is a subset of all sets belonging to greater lengths. This subset, however, is not necessarily a proper subset.

An appellation is said to be of the type and length, if any, of the value to which it refers. It is of structure zero and its level is one higher than the level of that value. If an appellation refers to a procedure, it is said to be of type procedure, structure zero and level one. Thus, the type procedure can only occur as an attribute of values with level greater than zero.

A reference may also be "free", i.e. the objects which it may refer to, are not restricted to a specific type, length or structure. It is of structure zero and its level is one higher than the level of that value. An assignment to a quantity to

which such a reference belongs, may only take place by applying the "conformity operator" within a conformity relation.

An appellation may refer to another appellation which itself may refer to another appellation, etc., until an object of level zero appears. In this way a "chain" of objects of decreasing level is given. The appellation of the highest level within a chain determines in which levels of this chain assignments may take place or not. The latter levels are called "constant levels". No assignment is possible to a quantity of such a constant level if it is accessed by means of appellations occurring in this chain. Within a chain no appellation belonging to a constant may appear on a nonconstant level.

An "array value" or "structured value" is a row of zero or more other values, each of which is of one same mode. It is of the type and length, if any, of these other values. Its structure is one higher than the structure of these other values, and it is of level zero. Since the other values may be array values themselves, the array value can be viewed as a one- or multi-dimensional rectangular array of values of one same unstructured mode.

The quantity to which an array value may belong, is called an array. In the case of an array, the above mentioned values of same unstructured mode belong to quantities, called "array elements". The array "contains" the array elements. The ordered set of the appellations belonging to these quantities is called the "common appellation" of the array, whereas the "value of the array" is the ordered set of the values of these quantities.

The array elements may be accessed by means of the "indexing operation".

A "record" is a value of a record type. It is given by an ordered set of "field values" which are not necessarily of the same mode.

The quantity to which a record may belong, is also called a record. Within such a quantity the field values also belong to

3.6. continued

quantities, called "fields". A record is said "to contain" its fields.

Fields may be accessed by means of an operation, called "field selection".

Records may be generated arbitrarily at all points in the program where the corresponding record type declaration is valid.

3.7. Correspondences of values

To each integral value of given length there corresponds a real value of that length and to each real value of given length there corresponds a complex value of that length. For the first case this is to mean, that in the class of reals of that length there is a value upon which all operations admissible to both values yield the same result as upon that integral value. An analogous definition applies to the second case as well.

To each binal value of given length there corresponds an integral value of that length, namely that nonnegative integral value which has the given sequence of binary digits as a possible value denotation in the binary number system.

Taking the value corresponding to a given value in the above sense, is an operation called "widening" of that value. For "narrowing" operations which are not always the inverse operations of widening operations see 7.19.

To each value of given nonboolean type and length there corresponds a value of the same type and next greater length, if such a greater length exists. This is simply a consequence of the fact that the sets of primitive values belonging to different lengths of one same nonboolean type form a sequence of subsets (cf. 3.6).

To each value of nonboolean type there belongs syntactically a certain number of long symbols. Taking the same value with a number of long symbols greater by one is an operation called

"lengthening" of that value. This does not necessarily mean that the length of the lengthened value is different from the length of the original value.

The inverse operation is called "shortening". Accordingly, this operation involves no action in case that the lengths specified by the two different numbers of long symbols are equal. Otherwise shortening of an integral, binal or alphameric value means taking the same value of the subset belonging to the same type with next smaller length. Shortening of a real or complex value means taking that value of the subset which is closest to the given value in the sense of proper rounding. These values do not always exist, if the subset is a proper subset.

3.8. Elementary actions

Actions whose decomposition into smaller actions is outside the realm of the language are called elementary actions.

Examples of elementary actions are: Taking the value or the appellation of a carrier; assigning a value to a variable; performing arithmetic operations; selecting a field value or a field variable; selecting a constituent expression; establishing a denotation; establishing a quantity; making a copy of a phrase or a "descriptor"; making modifications in a phrase.

3.9. Operations

"Monadic" and "dyadic" operations are mappings from one value and a pair of values respectively, the "operand(s)", into one value, the "result". Operations are denoted by "operators".

3.1o. The computation

By definition, the computation is the elaboration of the expression which is the program. At the beginning of that elaboration the computer contains the program and the "environment". This environment comprises the set of all primitive values occurring in the program as well as some quantities which are declared in so-called "environment-blocks", and which may be used freely during the elaboration of the program. During this elaboration, quantities are established and disestablished, and relationships are established and altered.

In the sequel, the syntax defines the internal structure that a sequence of symbols must have if it is to be a program. Under the heading Semantics the actions performed by the computer when elaboration of a phrase are defined. Both definitions are recursive.

In the language a specific denotation for values, identifiers and phrases is used which, together with the recursive definition of the language makes it possible to handle and to distinguish between arbitrarily long sequences of symbols, to distinguish between arbitrarily many different values of any nonboolean type, and to distinguish between arbitrarily many types; it allows further arbitrarily many objects to occur in the computer, and it allows the elaboration of a program to involve an arbitrarily large, not necessarily finite, number of actions.

This is not meant to imply either that the internal denotation of items in the computer is the denotation used in the language, or that it has the same possibilities. It is not assumed that these two denotations are the same or even that a one-to-one correspondence exists between them; in fact, the set of different internal denotations of objects of a given class may be finite,

3.1o. continued

and the number of classes for which this set is not empty may also be finite. It is, therefore, not assumed that the computer can handle arbitrary amounts of presented information. It is not assumed that the number of quantities and relationships that can be established is sufficient to cope with the requirements of a given program nor that the speed of operation of the computer is sufficient to elaborate a given program within a fixed lapse of time.

It is assumed, however, that these restrictions for classes of objects do not affect the effective presence of the features involved and touch only limitations, e.g., of reaches, or number of levels, of parameters, of dimensions, and of fields.

3.11. Implementations

A model of the hypothetical computer, using an actual machine, is said to be an "implementation" of the language, if it does not restrict the use of the language in other respects than those mentioned above. Furthermore, if additional restrictions are formulated, defining a language whose programs are a subset of the programs of the language, then that language is called a sublanguage of the language. 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.

4. Basic constituents of the language

A text, constituting a program, is a sequence of basic symbols. The notion of a basic symbol is introduced in this chapter. Moreover, methods are described for constructing certain denotations e.g. identifiers and numbers. In this and the following chapters, examples are given using the extensions and the representations for the basic symbols which are given in chapter 8.

4.1. Basic symbols

4.1.1. Syntax

* 4.1.1.1. basic symbol ::= ordinary basic symbol /
ordinary basic symbol, comment.

* 4.1.1.2. ordinary basic symbol ::= letter / value denotation
symbol / operator symbol / declaration
symbol / syntactical symbol / sequencing
symbol / extra symbol.

4.1.1.3. TOKEN sequence ::= TOKEN / TOKEN sequence, TOKEN.

4.1.1.4. comment ::= comment begin symbol, comment element
sequence, comment end symbol.

4.1.1.5. comment element sequence ::= character sequence.

4.1.1.6. character ::= proper character / other character.

4.1.1.7. string ::= quote symbol, string element sequence,
quote symbol / empty string.

4.1.1.8. string element sequence ::= character sequence.

4.1.1.9. empty string ::= quote symbol, quote symbol.

4.1.2. Semantics

A program is regarded as presented as a linearly ordered sequence of characters, running from left to right.

4.1.2. continued

Starting with the leftmost character, the characters of this sequence are grouped into a linearly ordered sequence of basic symbols also running from left to right.

In performing this process, typographical display characters such as blank space, change to a new line and change to a new page are irrelevant as long as they do not occur as constituents of strings.

This order is called the textual order, and in the sequel "following" is understood to mean "textually immediately following".

Characters which in this process may appear as constituents of representations of ordinary basic symbols, comment begin symbols, comment end symbols, or quote symbols are called "proper characters". In this connection they serve only to represent these symbols and have no meaning in themselves. Characters which enter the process as constituents of character sequences denote themselves. They are either proper characters or "other characters". The definition of the sets of proper characters and other characters is outside the realm of the language.

The existence of comments in the program does not affect the elaboration of the program. Comments only provide additional information to the human reader. The character sequence which represents the comment end symbol may not occur as part of a character sequence which serves as a comment element sequence. The character sequence representing the quote symbol may only enter a character sequence which serves as a string element sequence if there exists a specific rule about the correct matching of the closing quote symbol in this case. The definition of such rules is outside the realm of the language.

4.2. Letters

4.2.1. Syntax

4.2.1.1. letter ::= letter a symbol / letter b symbol /
letter c symbol / letter d symbol /
letter e symbol / letter f symbol /
letter g symbol / letter h symbol /
letter i symbol / letter j symbol /
letter k symbol / letter l symbol /
letter m symbol / letter n symbol /
letter o symbol / letter p symbol /
letter q symbol / letter r symbol /
letter s symbol / letter t symbol /
letter u symbol / letter v symbol /
letter w symbol / letter x symbol /
letter y symbol / letter z symbol /
other letter.

4.2.2. Semantics

Other letters are not specified. Letters are constituents of identifiers.

4.3. Value denotation symbols

4.3.1. Syntax

* 4.3.1.1. value denotation symbol ::= number symbol / truth symbol / binal symbol / string / neutral symbol.

* 4.3.1.2. number symbol ::= digit / point symbol / times ten to the power symbol.

4.3.1.3. truth symbol ::= true symbol / false symbol.

* 4.3.1.4. binal symbol ::= binary symbol / octal symbol / hexadecadic symbol / binary digit / octal digit / hexadecadic digit.

4.3.1.5. binary digit ::= zero symbol / one symbol.
4.3.1.6. octal digit ::= binary digit / two symbol / three symbol / four symbol / five symbol / six symbol / seven symbol.
4.3.1.7. digit ::= octal digit / eight symbol / nine symbol.
4.3.1.8. hexadecadic digit ::= digit / ten symbol / eleven symbol / twelve symbol / thirteen symbol / fourteen symbol / fifteen symbol.

4.3.2. Semantics

Value denotation symbols are constituents of value denotations. Digits are also constituents of identifiers. Octal digits and hexadecadic digits standing as part of octal numbers or hexadecadic numbers denote groups of three or four binary digits respectively. The conversion into such a sequence of binary digits is given by representing the octal or hexadecadic digit in the binary number system.

The neutral symbol denotes a value called the neutral value which is of any type, length and level but possesses structure zero.

4.4. Operator symbols

4.4.1. Syntax

* 4.4.1.1. operator symbol ::= dyadic plus symbol / dyadic minus symbol / times symbol / divided by symbol / integral divided by symbol / modulo symbol / to the power symbol / absolution symbol / plus i times symbol / monadic plus symbol / monadic minus symbol / equals symbol / differs symbol / conforms symbol / not conforms symbol/ conforms and becomes symbol / becomes symbol / less

symbol / not less symbol / greater symbol / not greater symbol / not symbol / and symbol / or symbol / left symbol / right symbol / concatenation symbol / tail symbol / head symbol / long symbol / short symbol / value symbol / name symbol binal to integral symbol / integral to real symbol / real to complex symbol / real part of symbol / imaginary part of symbol / round symbol / integral to binal symbol.

4.4.2. Semantics

Operator symbols are constituents of expressions. They denote operations on values or carriers. With the exception of the value operator and the name operator, the denoted operations have an inherent meaning which is independent of the language.

4.5. Declaration symbols

4.5.1. Syntax

* 4.5.1.1. declaration symbol ::= TYPE declaration symbol / array symbol / free symbol / constant symbol / constant procedure symbol / expression symbol / reference symbol / record type symbol / record symbol.

4.5.2. Semantics

Declaration symbols are constituents of declarations. They characterize relevant properties of entities which are to be declared.

4.6. Syntactical symbols

4.6.1. Syntax

* 4.6.1.1. syntactical symbol ::= open symbol / close symbol / if symbol / then symbol / else symbol/case symbol / of symbol / parallel open symbol / parallel close symbol / parallel separation symbol / sub symbol / up to symbol / at symbol / bus symbol / label symbol / comma symbol.

4.6.2. Semantics

Syntactical symbols are constituents of phrases, they serve to separate notions or group them together.

4.7. Sequencing symbols

4.7.1. Syntax

* 4.7.1.1. sequencing symbol ::= go on symbol / completion symbol / go to symbol.

4.7.2. Semantics

Sequencing symbols are constituents of phrases, they characterize the order im which different parts of a phrase are elaborated.

4.8. Extra symbols

4.8.1. Syntax

* 4.8.1.1. extra symbol ::= for symbol / from symbol / by symbol / to symbol / while symbol / except symbol / do symbol / fi symbol / repeat symbol.

4.8.2. Semantics

Extra symbols are constituents of phrases. They occur only in phrases which, by virtue of extensions, stand for phrases in which no extra symbols occur.

4.9. Identifiers

4.9.1. Syntax

4.9.1.1. identifier ::= letter / identifier, letter / identifier, digit.

4.9.1.2. ENTITY identifier ::= identifier.

4.9.2. Semantics

Identifiers have no inherent meaning but serve to identify "entities" i.e. variables, constants, record types, field selectors, and labels. About the determination of the entity which is identified by a given occurrence of an identifier see 5.8.2. and also 5.3.2.1.

The production 4.9.1.2. must be used in the following way: If an identifier occurs at a "defining occurence" (cf.5.8.2. and 6.1.3.), then that production of the metanotion "ENTITY" must be used which is " label " in case of a label definition or specified by the declarator preceding that identifier (cf.6.1.2.).

At any other occurence of an identifier a search must be made for the defining occurence of that identifier. If such a defining occurence exists, then the same production of the metanotion " ENTITY " must be used as at that defining occurence.

4.10. Numbers

4.10.1. Syntax

4.10.1.1. integral number ::= digit sequence.

4.10.1.2. signed integral number ::= integral number / inversion operator, integral number.

4.10.1.3. fractional part::= point symbol, integral number.

4.10.1.4. mantissa ::= integral number, fractional part / fractional part.

4.10.1.5. exponent ::= times ten to the power symbol, signed integral number.

4.1o.1.6. real number ::= mantissa / mantissa, exponent / integral number, exponent / exponent.

4.1o.1.7. binary number ::= binary symbol, binary digit sequence.

4.1o.1.8. octal number ::= octal symbol, octal digit sequence.

4.1o.1.9. hexadecadic number ::= hexadecadic symbol, hexadecadic digit sequence.

4.1o.1.1o. binal number ::= binary number / octal number / hexadecadic number.

4.1o.1.11. inversion operator ::= monadic plus symbol / monadic minus symbol.

4.1o.2. Semantics

Numbers have an inherent meaning which is acknowledged to exist independently of the language. They denote values of type integral, real, or binal respectively. Binal numbers denote sequences of the digits zero and one only. About the conversion of octal numbers and hexadecadic numbers in binary numbers see 4.3.2.

4.1o.3. Examples

```
12345          # (this is an integral number ) #
o.12345
.12
1o^-5
1.5_1o^2       # (these are real numbers      ) #
1.             # (this is not a number        ) #
b ooo1o1       # (this is a binary number     ) #
o 124          # (this is an octal number     ) #
h a8f53oc      # (this is a hexadecadic number) #
```

5. Program and phrases

This chapter introduces the overall structure of programs.
The notion of a program is defined by means of the notion of a block.
A block, in turn, is a grouping of one or more phrases. Note that a block is itself a primary (cf. 7.13.) and hence may appear as a phrase or a part of a phrase.
It is therefore the notion of a phrase which plays an important role in this chapter.

5.1. Program

5.1.1. Syntax

5.1.1.1. program ::= block.

5.1.2. Semantics

The program is an expression whose elaboration is by definition the "computation". It is assumed that the program is embedded in environment blocks which contain declarations of some entities which may occur in the program (cf. 3.1o.).

5.2. Phrases

5.2.1. Syntax

* 5.2.1.1. phrase ::= SORT expression / declaration.

5.2.1.2. SORT expression ::= proper SORT expression / conditional SORT expression / jump.

5.2.1.3. proper SORT expression ::= SORT assignment / plain SORT expression.

5.2.1.4. admissible constant ORDINARY expression ::= ORDINARY expression.

5.2.1.5. admissible RAISED constant NONCONSTANT expression ::= RAISED NONCONSTANT expression.

5.2.1.6. adjusted SORT expression ::= SORT expression / admissible SORT expression / adjusted constant SORT expression.

5.2.1.7. free expression ::= FULL expression.
5.2.1.8. constant free expression ::= FULL expression / constant FULL expression.
5.2.1.9. RAISED free expression ::= RAISED FULL expression.
5.2.1.1o. RAISED constant free expression ::= RAISED constant FULL expression.
5.2.1.11. block ::= SORT block.
5.2.1.12. SORT block ::= unitary SORT block / neutral expression.
5.2.1.13. neutral expression ::= neutral symbol / parallel block.

5.2.2. Semantics

The notion of an admissible expression is used in conditional expressions (cf. 5.5.) and array denotations (cf.7.8.) showing the fact that a chain of objects in which certain levels are not constant may also be used as if these levels were constant. (cf. 3.6.).

Adjusted expressions are used in describing initialized declarations (cf. 6.3.) and actual parameters in procedure designators (cf. 7.12.).

The elaboration of a phrase begins when it is "initiated".
It may be "interrupted", and it is finished by being either "terminated" or "completed".
Upon finishing the elaboration of a phrase, either the computation is finished (in this case the phrase was the whole program) or a "successor" may be appointed (for the definition of successor cf.5.7.).

5.2.2.1. Interruption of the elaboration of a phrase

The elaboration of a phrase may be interrupted by an action, e.g. overflow, not specified by that phrase but taken by the computer if its possibilities and limitations do not permit a correct computation.

5.2.2.1. continued

Whether after an interruption the elaboration of that phrase is resumed,or the elaboration of another phrase is initiated,or the computation is finished,is a question outside the realm of the language.

5.2.2.2. Termination of the elaboration of a phrase

The elaboration of a phrase is terminated by the elaboration of a jump (cf.5.6.).
If there exist other phrases which are elaborated together with the given phrase,either in parallel or in unspecified order,then these other phrases are also terminated (cf.5.4.,7.1.,7.2.,7.8.,7.10.,7.12.,7.16.).
Upon termination of a phrase the elaboration of the succeeding phrase which is appointed by the jump is initiated.
An expression which is terminated possesses no value.

5.2.2.3. Completion of the elaboration of a phrase

If there occurs no interrupt and no jump during the elaboration of a phrase,then the elaboration of that phrase is completed after performing the last action specified by that phrase.
A phrase being an expression possesses a value upon completion.
If the completed phrase appoints a successor,then the elaboration of that successor is initiated.

The elaboration of the expression denoted by the neutral symbol is always completed.It consists of taking the neutral value as the value of this expression.

5.2.3. Example

```
nil     # (this is the neutral expression
          denoted by the neutral symbol) #
```

5.3. Unitary blocks

5.3.1. Syntax

5.3.1.1. unitary SORT block ::= blockhead,SORT expression part,
close symbol.

5.3.1.2. blockhead ::= open symbol / open symbol,declaration part,
go on symbol.

5.3.1.3. declaration part ::= single declaration / expression
declaration part.

5.3.1.4. expression declaration part ::= PROCEDURE declaration /
expression declaration part,
go on symbol,
PROCEDURE declaration.

* 5.3.1.5. declaration ::= single declaration / PROCEDURE declaration.

5.3.1.6. SORT expression part ::= SORT expression series /
SORT expression part,completer,
SORT expression series.

5.3.1.7. completer ::= completion symbol,label definition.

5.3.1.8. label definition ::= label identifier,label symbol.

5.3.1.9. SORT expression series ::= SORT expression / statement,
go on symbol,SORT expression
series / label definition,
SORT expression series.

5.3.1.10. statement ::= proper SORT expression / conditional statement / jump.

* 5.3.1.11. neutral block ::= blockhead,neutral expression part,
close symbol.

* 5.3.1.12. neutral expression part ::= neutral expression series /
neutral expression part, completer,
neutral expression series.

* 5.3.1.13. neutral expression series ::= neutral expression / state-
ment, go on symbol, neutral expression
series / label definition, neutral
expression series.

5.3.2. Semantics

The notion neutral expression is used in 6.4.3.

5.3.2.1. Initiation of unitary blocks

The elaboration of a unitary block is initiated by performing the following steps:

Step 1: If an identifier occurs within the unitary block which identifies within the block a different entity than it does at the place from which the elaboration is initiated, then it is replaced throughout the unitary block by an identifier which is defined neither within the unitary block nor at that place, and step 1 is taken again, otherwise step 2 is taken.

Step 2: If a declaration part exists, then it is elaborated, otherwise step 3 is taken.
Upon completion of the elaboration of the declaration part, step 3 is also taken.
For the termination of the elaboration of the declaration part see 6.0.2.

Step 3: The elaboration of the first expression following the blockhead is initiated.

If a declaration part consists of more than one declaration, then these declarations are elaborated in unspecified order.
For the elaboration of declarations see chapter 6.

5.3.2.2. Completion of the elaboration of a unitary block and its value.

The elaboration of a unitary block is completed upon the completion of the elaboration of the last expression of one of the expression series which constitute the expression part. The value of the completed unitary block is the value of the last elaborated expression.

5.3.2.2. continued

If this value is a reference, an array of references or a record containing references, then a search to determine the range of all references contained in the value is made in the same way as in 5.8.2. If one of these references has a range identical with the block which is now completed, then the further elaboration is undefined.

5.3.3. Extensions

a) In the situations "single declaration, open symbol, declaration part, go on symbol, expression part, close symbol, close symbol" or "declaration part, open symbol, single declaration, expression part, close symbol", the open symbol and the corresponding close symbol may be omitted.

Remark: This extension allows one to write more than one single declaration in one blockhead.The rules for identification (cf. 5.8.2.) are, of course, not affected by this extension.

b) If in the situations "actual CARRIER declarator, CARRIER identifier, CARRIER initialization, go on symbol, actual CARRIER declarator, CARRIER identifier" or "actual CARRIER declarator, CARRIER identifier, go on symbol, actual CARRIER declarator, CARRIER identifier" the actual carrier declarators are identical, then the go on symbol may be replaced by "comma symbol" and the second declarator may be omitted (for declarators cf.6.1.).

c) If the last expression of an expression series (which is therefore followed by a completer or a close symbol) consists only of a neutral symbol, then this neutral symbol may be omitted.

5.3.4. Examples

```
begin real a,b;  a:=2×b  end
# (which is short for:
begin real a;
      begin real b;  a:=2×b  end
end )#
```

5.4. Parallel blocks

5.4.1. Syntax

5.4.1.1. parallel block ::= parallel open symbol,block,parallel blocktail.

5.4.1.2. parallel blocktail ::= parallel separation symbol, block,parallel close symbol / parallel separation symbol, block,parallel blocktail.

5.4.2. Semantics

The elaboration of a parallel block is initiated by initiating the elaboration of all constituent blocks in parallel (cf.3.1.). It is terminated upon termination of the elaboration of one of the constituent blocks.It is completed upon the completion of the elaboration of all constituent blocks.In this case the value of the parallel block is by definition the neutral value.

5.4.3. Remark

Note that the language does not prohibit the use of a value within a constituent block which is possibly changed by another constituent block.

5.5. Conditional expressions

5.5.1. Syntax

5.5.1.1. if clause ::= if symbol,boolean expression,then symbol.

5.5.1.2. conditional SORT expression ::= if clause,SORT expression,else symbol,SORT expression /
if clause,admissible SORT expression,else symbol,SORT expression /
if clause,SORT expression,else symbol,admissible SORT expression.

5.5.1.3. conditional statement ::= if clause, expression, else symbol, expression.
5.5.1.4. expression ::= SORT expression.

5.5.2. Semantics

The elaboration of a conditional expression is performed in the following steps:

Step 1: The boolean expression is elaborated.If this elaboration is completed,then step 2 is taken,otherwise step 4 is taken.

Step 2: If the value of the boolean expression is true,then the expression immediately following the then symbol is selected;otherwise the expression immediately following the else symbol is selected.

Step 3: The selected expression is elaborated.If this elaboration is completed,then the value of this expression is the value of the conditional expression and the elaboration of the conditional expression is completed.

Step 4: The elaboration of the conditional expression is terminated.

5.5.3. Extensions

If the expression following the else symbol consists of the neutral symbol only,then the sequence "else symbol,neutral symbol" may be replaced by "fi symbol".

5.5.4. Examples

if x ∨ a ∧ b then x:=false fi

if c::=d then d:=name x else x:=5

5.6. Jump

5.6.1. Syntax

5.6.1.1. jump ::= goto symbol,label identifier.

5.6.2. Semantics

A jump is an expression whose elaboration is never completed, but always terminated.It appoints as its successor the expression which immediately follows the definition of the constituent label identifier.The elaboration of all expressions which are contained in the smallest unitary block in which that label definition appears,is also terminated.

5.7. Determination of successors

A phrase appoints its successor in the following cases:

Case 1: A jump appoints as its successor the expression immediately following the definition of the label which appears in that jump (cf.5.6.1.).

Case 2: A phrase whose elaboration is terminated appoints as its successor,the successor of that jump whose elaboration has terminated the elaboration of that phrase.

Case 3: If among the phrases which are elaborated in an unspecified order there is a phrase whose elaboration is terminated,then all other phrases are also terminated and the appointed successor is the successor

5.7. continued

of that phrase whose elaboration caused the termination.

Case 4: Upon completion of the elaboration of an expression which is followed by a go on symbol, the expression following the go on symbol is appointed as successor.

Case 5: Upon completion of the elaboration of an expression which is followed by a completion symbol, the expression which is the successor of the smallest block containing that completion symbol is appointed as successor, provided this block appoints a successor.

In all other cases no successor is appointed.

5.8. Identification of entities

5.8.1. Syntax

» 5.8.1.1. definition :: = label definition / declaration.

5.8.2. Semantics

An entity has one, and only one, definition. This definition associates an identifier with that entity and there is within this definition a "defining occurrence" of that identifier. This defining occurrence is also called the defining occurrence of that entity. In the case of a label identifier the defining occurrence is the occurrence of that identifier within the label definition.
For defining occurrences of identifiers in declarations see 6.1.3.
If during the elaboration of the program an identifier occurs and this occurrence is not a defining occurrence, then the defining occurrence of the identifier may be found by the following process:

Step 1: Call the smallest expression which contains the given occurrence of the identifier the "possible range" of this identifier.

5.8.2. continued

Step 2: Take the smallest unitary block which contains the possible range and call now this block the possible range.If there exists no such embracing block in the program,then step 4 is taken.If this block contains a defining occurence of the given identifier which is not contained in any smaller block,then step 3 is taken.Otherwise step 2 is repeated.

Step 3: If the given block contains more than one defining occurence of this identifier,then the further elaboration is undefined.Otherwise at the given occurence that identifier identifies the entity defined by the definition. The possible range is called the "range of this entity" and the search is finished.

Step 4: If there exists a defining occurence of that identifier in some embracing block belonging to the environment (cf.3.1o.,5.1.2.),then at the given occurence the identifier identifies the entity defined by that definition. Otherwise the identifier possesses no defining occurence and the further elaboration is undefined.

6. Declarations

In this chapter methods for declaring quantities and record types are introduced. Declarations serve either to establish quantities or at least to prepare for this action (in case of record types and records).

6.0.1. Syntax

* 6.0.1.1. declaration ::= single declaration /
PROCEDURE declaration.

6.0.1.2. single declaration ::= actual CARRIER declaration /
RECORD type declaration.

6.0.2. Semantics

A declaration establishes an entity and causes an identifier to denote this entity.

For the elaboration of procedure declarations see 6.4., of carrier declarations see 6.3., and of record type declarations see 6.2..

If, during the elaboration of an expression contained within a single declaration, a jump is elaborated which appoints as successor an expression outside the declaration but within the smallest block containing the declaration, then the further elaboration is undefined.

If a carrier declaration defines an identifier and this identifier occurs in any expression contained in this declaration, then the further elaboration is undefined.

An entity established by the elaboration of a declaration ceases to exist upon termination or completion of the elaboration of the smallest block containing that declaration.

6.1. Declarators

6.1.1. Syntax

* 6.1.1.1. declarator ::= ACTAL CARRIER declarator / RECORD type declarator / ACTAL PROCEDURE declarator.

6.1.1.2. ACTAL TYPE declarator ::= TYPE declaration symbol.

6.1.1.3. ACTAL long NONBOOLEAN declarator ::= long symbol, ACTAL NONBOOLEAN declarator.

6.1.1.4. ACTAL free declarator ::= free symbol.

6.1.1.5. ACTAL procedure with a MODE result and an undetermined number of parameters declarator ::= ACTAL procedure with a MODE result declarator.

6.1.1.6. ACTAL procedure with a MODE result and PARAMETERS declarator ::= ACTAL procedure with a MODE result declarator.

6.1.1.7. ACTAL procedure with a MODE result declarator ::= constant procedure symbol, actual MODE declarator.

6.1.1.8. ACTAL reference to a SPECIAL declarator ::= reference symbol, ACTAL SPECIAL declarator.

6.1.1.9. ACTAL reference to a constant free declarator ::= reference symbol, ACTAL constant free declarator.

6.1.1.10. ACTAL reference to a CARRIER declarator ::= reference symbol, ACTAL CARRIER declarator.

6.1.1.11. ACTAL constant NONCONSTANT declarator ::= constant symbol, ACTAL NONCONSTANT declarator.

6.1.1.12. ACTAL array of ORDINARY declarator ::= array symbol, ACTAL boundpair, ACTAL ORDINARY declarator.

6.1.1.13. ACTAL boundpair ::= sub symbol, ACTAL lower bound, up to symbol, ACTAL upper bound, bus symbol.

6.1.1.14. formal LOWPER bound ::= strict LOWPER bound / flexible LOWPER bound.

6.1.1.15. actual LOWPER bound ::= strict LOWPER bound.

6.1.1.16. strict LOWPER bound ::= integral expression.

6.1.1.17. flexible LOWPER bound ::= constant symbol, integral symbol, constant integral identifier.

6.1.1.18. ACTAL RECORD declarator ::= record symbol, open symbol, ACTAL RECORD type indicator, close symbol.

6.1.1. continued

6.1.1.19. RECORD type declarator ::= record type symbol,
RECORD indicator.
6.1.1.2o. actual RECORD type indicator ::= RECORD type identifier.
6.1.1.21. formal RECORD type indicator ::= RECORD type identifier
RECORD indicator, RECORD type identifier.
6.1.1.22. RECORD indicator ::= open symbol, proper RECORD indicatc
close symbol.
6.1.1.23. proper record with a ORDINARY field indicator ::=
ORDINARY field declaration.
6.1.1.24. proper RECORD and a ORDINARY field indicator ::=
proper RECORD indicator, separator, ORDINARY field
declaration.
6.1.1.25. ORDINARY field declaration ::= actual ORDINARY declaratc
ORDINARY field identifier.
6.1.1.26. separator ::= comma symbol / close symbol, open symbol.

6.1.2 Semantics

Actual declarators serve for specifying certain attributes which are associated with entities which are established either by the declaration containing those actual declarators or by record generators appearing in the further elaboration. In case of arrays and records actual declarators serve also for constructi descriptors for these quantities.

Record type declarators serve for establishing a new record type and reserve identifiers for further use as field identifiers.

Formal declarators occur only in the formal parameter part of procedure torsos. They serve for constructing actual declarators during the elaboration of procedure designators (cf. 7.12.).

The mentioned set of attributes associated with a quantity is syntactically mirrored by those parts of the notions of the strict language which are productions of the metanotions "CARRIER" and "PROCEDURE". These parts of notions generate also terminal symbols in declarators.

6.1.3. Defining occurrences

A given occurrence of an identifier is a defining occurrence of that identifier if one of the following conditions is satisfied:

a) The identifier is immediately preceded by an actual or formal declarator and does not stand as a field identifier.

b) The identifier is a constituent of a flexible lower or upper bound.

c) The identifier occurs as a constituent field identifier of a field declaration which is itself not part of a formal record type indicator.

6.1.4. Extensions

a) The sequence "bus symbol, array symbol, sub symbol" may be replaced by "bus symbol, sub symbol".

b) The sequence "bus symbol, sub symbol" may be replaced by "comma symbol".

c) The sequence "constant symbol, integral symbol" may be replaced by "integral symbol" when occurring as part of a flexible bound.

d) If no further use is made of the identifier occurring in a flexible bound, then this flexible bound may be omitted.

e) The sequence "close symbol, open symbol" may be replaced by a sequence starting with "close symbol" followed by an arbitrary sequence of letters and/or digits and ending with "open symbol".

6.2. Record type declarations

6.2.1. Syntax

6.2.1.1. RECORD type declaration ::= RECORD type declarator, RECORD type identifier.

6.2.2. Semantics

A record type declaration causes the constituent identifier to denote a new record type.Also it reserves some identifiers for further use as field identifiers.

The elaboration of a record type declaration involves no action,especially the field declarations contained in the record indicator are not elaborated and no field selectors are established (but cf.7.1o.).

6.2.3. Example

rec type (compl algol x,ref real aad,rec (wg21) tony) wg21

6.3. Carrier declarations

6.3.1. Syntax

* 6.3.1.1. CARRIER declaration ::= ACTAL CARRIER declaration.
6.3.1.2. initialized CARRIER declaration ::= actual CARRIER declarer, equals symbol,CARRIER initialization.
6.3.1.3. ACTAL CARRIER declarer ::= ACTAL CARRIER declarator, CARRIER identifier.
6.3.1.4. CARRIER initialization ::= adjusted CARRIER expression / adjusted constant CARRIER expression.
6.3.1.5. actual constant ORDINARY declaration ::= initialized constant ORDINARY declaration.
6.3.1.6. actual PRIMITIVE declaration ::= actual PRIMITIVE declarer / initialized PRIMITIVE declaration.
6.3.1.7. actual ARRAY PRIMITIVE declaration ::= actual ARRAY PRIMITIVE declarer / initialized ARRAY PRIMITIVE declaration.
6.3.1.8. actual RECORD declaration ::= initialized RECORD declaration
6.3.1.9. actual ARRAY RECORD declaration ::= initialized ARRAY RECORD declaration.

6.3.1. continued

6.3.1.10. actual LEVELED declaration ::= initialized LEVELED declaration.

6.3.1.11. actual ARRAY LEVELED declaration ::= initialized ARRAY LEVELED declaration.

6.3.1.12. formal CARRIER declaration ::= formal CARRIER declarer.

6.3.2. Semantics

For the use of formal declarations see 6.4..

The elaboration of an actual carrier declaration proceeds as follows:

Step 1: If in the declarator there occur bound pairs and/or there exists an initialization, then the expression contained in the boundpairs and/or the adjusted expression in the initialization are elaborated in unspecified order.

Step 2: A quantity is created with the mode specified by the declarator. The identifier following the declarator is made to denote this quantity.

In the case of an array a descriptor is established (cf. 6.3.3.) and associated with this quantity. Further the ordered set of quantities constituting this array is created.

Step 3: If there exists an initialization, then the result of the adjusted expression is assigned to the created quantity (cf. 7.1. and 7.4.2.,conformity), otherwise the value of the created quantity is said to be undefined.

If in the case of a record carrier the initialization yields a record, then only the descriptor of that record is associated with the created quantity (cf. 7.1.2., step 4), otherwise the value of the quantity is the neutral value.

6.3.3. Descriptors for arrays

A descriptor for an array is a technical device for selecting array elements or groups of array elements (cf. 7.16.). It consists of an integral number n, called the "dimension", an integral number c, called the origin, and an ordered set of triples (l_i, u_i, d_i), $i = 1, \ldots, n$.

The array descriptor is constructed from an array declarator (cf. 6.1.1.11., 6.1.1.12.) in the following way.

Step 1: Without regard to a possible constant symbol in front of the declarator, the array declarator begins with a number of array symbols each followed by a bound pair. This sequence is followed by a declarator which is not an array declarator. The dimension n in the descriptor is set equal to the number of array symbols mentioned above. The origin c is set equal to 1, and a dimension pointer j is set also equal to 1. The bound pairs contained in the declarator are ordinally numbered from one upwards, running from left to right.

Step 2: The triple (l_j, u_j, d_j) corresponding to the current value of the dimension pointer is initialized with $d_j = 1$, l_j equal to the value of the lower bound and u_j equal to the value of the upper bound of the bound pair corresponding to the current value of j. For $i = 1, \ldots, j-1$, d_i is multiplied with $(u_j - l_j + 1)$.

Step 3: j is increased by 1, and step 2 is taken again if $j \leq n$.

6.3.4. Descriptors for records

A record descriptor is created during the elaboration of a record generator (cf. 7.1o.). It consists of an ordered set of "field selectors" and an ordered set of identifiers denoting these field selectors. These sets are obtained as follows:

Step 1: The record type identifier of the record generator has a defining occurrence in a record type declaration. The record indicator of this declaration contains a set of field declarations. These field declarations are elaborated in unspecified order as if they were actual declarations which are not initialized. The ordered set of quantities resulting from this elaboration forms a new record.

Step 2: A set of field selectors is constructed which allows the selection of the constituent quantities from the record. The constituent field identifiers of the record indicator are made to denote the corresponding field selectors.

6.3.5. Extensions

In case of records, leveled quantities, or arrays of such quantities, an initialization of the form "equal symbol, neutral symbol" or "equal symbol, constant STRUCTURED direct denotation" in which all expressions yielding values of structure zero consist of the neutral symbol only, may be omitted.

6.3.6. Examples

rec (wg21) munich = nil

6.4. Procedure declaration

6.4.1. Syntax

6.4.1.1. PROCEDURE declaration ::= actual PROCEDURE declaration.

6.4.1.2. actual PROCEDURE declaration ::= actual PROCEDURE declarer, equals symbol, expression symbol, PROCEDURE torso.

6.4.1.3. formal PROCEDURE declaration ::= formal PROCEDURE declarer.

6.4.1.4. ACTAL PROCEDURE declarer ::= ACTAL PROCEDURE declarator, PROCEDURE identifier.

6.4.1.5. procedure with a MODE result torso ::= adjusted MODE expression.

6.4.1.6. procedure with a MODE result and PARAMETERS torso ::= open symbol, formal PARAMETERS, close symbol, go on symbol, ad~~missible~~ justed MODE expression.

6.4.1.7. formal CARRIER parameter ::= formal CARRIER declaration.

6.4.1.8. formal PROCEDURE parameter ::= formal PROCEDURE declaration.

6.4.1.9. formal PARAMETERS and a PARAMETER ::= formal PARAMETERS, separator, formal PARAMETER.

* 6.4.1.10. procedure body ::= open symbol, PROCEDURE torso, close symbol.

6.4.2. Semantics

The elaboration of a procedure declaration is performed in the following way:

A quantity is created which consists of an appellation and the procedure torso which is made into a procedure body by inserting an open symbol in front of it and a close symbol after it. Then an identifier is caused to denote this quantity.

6.4.3. Extensions

a) The sequence "equals symbol, expression symbol" may be omitted.

b) In the situation "constant procedure symbol, actual MODE declarator, PROCEDURE identifier, equals symbol, PROCEDURE ~~symbol~~ torso" the actual mode declarator may be omitted if the constituent expression of the procedure torso is a neutral block or a neutral expression (cf. 5.2., 5.3.).

c) In the situation "constant procedure symbol, formal MODE declarator, PROCEDURE identifier" the formal mode declarator may be omitted if each actual declaration constructed from this formal declaration has a procedure torso whose constituent expression is a neutral block or a neutral expression.

d) In the situation "reference symbol, ACTAL procedure with a MODE result and an undetermined number of parameters declarator", the "ACTAL MODE declarator" contained in the given declarator may be omitted, if the declared quantity refers only to procedures which have a neutral block or a neutral expression as constituent expression of their torso. Note that the declarator in question may be preceded by other declaration symbols.

e) If in the situations "formal CARRIER declarator, CARRIER identifier, comma symbol, formal CARRIER declarator" or " formal PROCEDURE declarator, PROCEDURE identifier, comma symbol, formal PROCEDURE declarator" the two declarators are textually identical, then the second one may be omitted.

6.4.4. Examples

For examples see chapter 12.

7. Proper expressions

A proper expression is either an assignment or a plain expression. A plain expression, if not of primitive mode, is a primary. An assignment in which a quantity or value of free kind occurs is only possible in connection with a conformity relation (cf. 7.4.).

7.1. Assignments

7.1.1. Syntax

7.1.1.1. NONFREE assignment ::= NONFREE destination, becomes symbol, NONFREE source.

7.1.1.2. NONFREE source ::= NONFREE expression.

7.1.1.3. SORT destination ::= SORT carrier identification / SORT reference identification / admissible SORT destination.

7.1.1.4. NONCONSTANT carrier identification ::= NONCONSTANT identifier / NONCONSTANT field selection / indexed NONCONSTANT primary.

7.1.1.5. NONCONSTANT reference identification ::= unitary reference to a NONCONSTANT block / reference to a NONCONSTANT procedure designator / reference to a NONCONSTANT case expression / evaluated NONCONSTANT primary.

7.1.1.6. admissible constant NONCONSTANT destination::= NONCONSTANT destination.

7.1.1.7. admissible RAISED SORT destination ::= RAISED constant SORT destination.

7.1.2. Semantics

The elaboration of an assignment proceeds as follows:

Step 1: The destination and the source are elaborated in an unspecified order. The result of the elaboration of

7.1.2. continued 1

the destination is an appellation (see below). The quantity to which this appellation belongs is called the "target quantity".

Step 2: If the value of the source is an array or a constant array, then step 3 is taken. If it is of a record type, then step 4 is taken. If it is an appellation, then step 5 is taken. Otherwise the value of the target quantity is replaced by the value of the source and step 6 is taken.

Step 3: The descriptors of the target quantity and of the value of the source are compared. If the lower and upper bounds, or the dimensions contained in these descriptors are not identical, then the further elaboration is undefined. If the array elements constituting the source are references, then the check described in step 5 is made for each array element vs. the target quantity. Then the constituent values of the source and the quantities forming the target quantity are ordinally numbered from one upwards and the values of these quantities are replaced in unspecified order by the values of the source with equal ordinal numbers. Step 6 is taken.

Step 4: If the value of the source is the neutral value then the value of the target quantity is replaced by the neutral value. Otherwise, if some of the fields constituting the source are references, then the check described in step 5 is made for these fields vs. the target quantity. Then the descriptor of the record is associated with the target quantity whereby all fields of the record are now accessible by field selections using the target quantity. Step 6 is taken.

Step 5: A search is made for determining the ranges of the target quantity and of the value of the source in the same way as in 5.8.2. If the range of the target quantity is contained in the range of the value of the source, then the value of the target quantity is replaced by the value of the source and step 6 is taken. Otherwise the further elaboration is undefined.

Step 6: The elaboration of the assignment is completed and the value of the source is also the value of the assignment.

The appellation which is the value of the destination is obtained as follows:

If the destination is a reference identification, then the appellation to be determined is the value of that reference identification. If this value is the neutral value, then the further elaboration is undefined.

If the destination is an identifier, a field selection, or an indexed primary, then the appellation to be determined is the appellation which belongs to the quantity given by this identifier, field selection, or indexed primary.

If the destination is an evaluated primary, then the appellation to be determined is the value resulting from the elaboration of that primary without regard to the value symbol in front of it. If this value is the neutral value, then the further elaboration is undefined.

7.1.3. Extensions

a) If the source is a named primary, then the name symbol in front of it may be omitted.
b) If the source is a lengthened primary, then the long symbol in front of it may be omitted.
c) If the source is of the form "long symbol, open symbol, plain NONBOOLEAN expression, close symbol", then it may be replaced by "plain NONBOOLEAN expression".

7.1.4. Examples

```
(if a < b  then name x real else name y real) := 3.14
d ref real := name x real
#(the occurring identifiers are chosen to denote their
  modes intuitively)#
```

7.2. Plain expressions

7.2.1. Syntax

7.2.1.1. plain PRIMITIVE expression ::= simple PRIMITIVE expression.

7.2.1.2. plain NONPRIMITIVE expression ::= NONPRIMITIVE primary.

7.2.2. Semantics

A simple expression other than a primary consists either of a monadic operator followed by an operand or of two operands separated by a dyadic operator.

The elaboration of such an expression is performed in two steps:

Step 1: If there are two operands, then these are elaborated in unspecified order (for an exception see 7.4.2., disjunction and conjunction). If there is only one operand, then it is elaborated.

Step 2: The operation denoted by the operator is performed on the value(s) obtained in step 1, yielding the value of the expression.

If an operand itself is not a primary, then it has the same form as a simple expression and its elaboration is performed in the same way.

For simple arithmetic expressions see 7.3., for simple boolean expressions see 7.4., for simple binal expressions see 7.5., for simple alphameric expressions see 7.6., for primaries see 7.7.

7.3. Simple arithmetic expressions

7.3.1. Syntax

7.3.1.1. simple ARITHMETIC expression ::= ARITHMETIC term/ inversion operator, ARITHMETIC term/ simple ARITHMETIC expression, addition operator, ARITHMETIC term.

7.3.1.2. ARITHMETIC term ::= ARITHMETIC factor/ ARITHMETIC term, ARITHMETIC multiplication operator, ARITHMETIC factor.

7.3.1.3. ARITHMETIC factor ::= ARITHMETIC secondary/ ARITHMETIC factor, exponentiation operator, amount.

7.3.1.4. LONG complex secondary ::= LONG complex primary/ LONG real part, pair operator, LONG imaginary part.

7.3.1.5. LONG real part ::= LONG real primary.

7.3.1.6. LONG imaginary part ::= LONG real primary.

7.3.1.7. LONG real secondary ::= LONG real primary/ absolution operator, LONG real primary/ absolution operator, LONG complex primary.

7.3.1.8. INTEGRAL secondary ::= INTEGRAL primary/ absolution operator, INTEGRAL primary.

7.3.1.9. inversion operator ::= monadic plus symbol/ monadic minus symbol.

7.3.1.10. addition operator ::= dyadic plus symbol/ dyadic minus symbol.

7.3.1.11. NONINTEGRAL multiplication operator ::= times symbol/ divided by symbol.

7.3.1.12. INTEGRAL multiplication operator ::= times symbol/ integral divided by symbol/ modulo symbol

7.3.1.13. exponentiation operator ::= to the power symbol

7.3.1.14. pair operator ::= plus i times symbol.

7.3.1.15. absolution operator ::= absolution symbol.

7.3.1.16. amount ::= integral primary.

7.3.2. Semantics

The dyadic plus symbol denotes addition, the dyadic minus symbol subtraction.

The times symbol denotes multiplication, the divided by symbol division without remainder.

The integral divided by symbol and the modulo symbol denote the operations yielding quotient and remainder respectively, resulting from a division with remainder. These operations are defined in terms of the language (see 7.3.4.).

The to the power symbol denotes exponentiation. If the amount is negative, then the result of the exponentiation is undefined.

The plus i times symbol denotes the pairing of two real values to a complex value.

The monadic plus symbol denotes no operation, the monadic minus symbol sign inversion.

The absolution symbol denotes taking the absolute value.

If the operand(s) required in connection with these operations possess integral values, then these operations are understood in the ordinary mathematical sense. If at least one operand is a real or complex value, then these operations are understood in the sense of numerical analysis, i.e. yielding a value possibly slightly deviating from the mathematically defined result of the operation performed.

7.3.3. Examples

```
x + y ↑ 2
3 × (a mod b) - c[i - 1]
z + (abs x) i (-1)× x i u[j]
```

7.3.4. Remark

The operations denoted by the integral divided by symbol and the modulo symbol are defined by the following procedures, the second using the first. For integers of greater length the operations are defined accordingly.

```
proc int integral divided by (const int a, b);
        if b = o then goto zerodivision in the environment else
        (if a×b < o then -1 else 1) ×
        (int i = o, j = abs a; repeat: if j < abs b then i else
        (j := j - abs b; i := i+1; goto repeat))

proc int modulo (const int a, b);
        a - integral divided by (a, b)×b
```

7.4. Simple boolean expressions

7.4.1. Syntax

7.4.1.1. simple boolean expression ::= conjunction/ simple boolean expression, or operator, conjunction.

7.4.1.2. conjunction ::= negation / conjunction, and operator, negation.

7.4.1.3. negation ::= boolean ternary/ not operator, boolean ternary.

7.4.1.4. boolean ternary ::= boolean secondary/ equality relation.

7.4.1.5. equality relation ::= simple NONBOOLEAN expression, equality operator, simple NONBOOLEAN expression/ boolean primary, equality operator, boolean primary/ LEVELED primary, equality operator, LEVELED primary.

7.4.1.6. boolean secondary ::= boolean primary/ conformity relation / order relation.

7.4.1.7. order relation ::= simple NONCOMPLEX expression, order operator, simple NONCOMPLEX expression.

7.4.1.8. conformity relation ::= GRADED FULL destination, conformity operator, GRADED full primary/ GRADED constant FULL destination, conformity operator, GRADED constant full primary/ FULL destination, conformity operator, full primary.
7.4.1.9. GRADED full primary ::= GRADED FULL primary.
7.4.1.10.GRADED constant full primary ::= GRADED constant FULL primary.
7.4.1.11.full primary ::= FULL primary/ constant FULL primary.
7.4.1.12.or operator ::= or symbol.
7.4.1.13.and operator ::= and symbol.
7.4.1.14.not operator ::= not symbol.
7.4.1.15.equality operator ::= equals symbol/ differs symbol.
7.4.1.16.order operator ::= less symbol/ not less symbol/ greater symbol/ not greater symbol.
7.4.1.17.conformity operator ::= conforms symbol/ not conforms symbol/ conforms and becomes symbol.

7.4.2. Semantics

If applied to boolean operands, the or symbol denotes disjunction, the and symbol conjunction.

The disjunction is elaborated as follows: The first operand is elaborated. If its value is true, then it is the result of the disjunction, otherwise the second operand is elaborated and its value is the result of the disjunction.

The conjunction is elaborated as follows: The first operand is elaborated. If its value is false, then it is the result of the conjunction, otherwise the second operand is elaborated and its value is the result of the conjunction.

If applied to a boolean operand the not symbol denotes negation.

The equals symbol and the differs symbol denote equality and unequality respectively, i.e. the result is true if the values of the two operands are identical or not identical respectively, and false otherwise.

The application of an order operator yields true if the sizes of the values of the two operands stand in the indicated relation, false otherwise.

The operation denoted by the conforms symbol yields true if the value of the second operand is "assignment compatible" (see below) with the quantity to which the appellation resulting from the destination belongs; otherwise the operation yields false. If the value of the destination is the neutral value, the conforms operation is undefined. If the value of the second operand is the neutral value, the conforms operation yields true.

The operation denoted by the not conforms symbol yields a result which is false if the conforms operation yields true and vice versa.

The operation denoted by the conforms and becomes symbol yields the same result as the conforms operation. Moreover, the value of the second operand is assigned to the target quantity according to the rules given in 7.1.2. For the elaboration of a destination see also 7.1.2.

A value and a quantity are called assignment compatible if the value may be assigned to that quantity according to the rules given in 7.1. or if the mode of the quantity is graded free and the value is of any mode but has the same level.

7.4.3. Extension

If the primary following a conformity operator is a named primary, then the name symbol in front of it may be omitted.

7.4.4. Examples

```
x real  ::= val  f ref free
val f ref free ::= x real
a ∀ b∧a≠b
name x = name y
```

7.5. Simple binal expressions

7.5.1. Syntax

7.5.1.1. simple BINAL expression ::= BINAL term / simple BINAL expression,or operator,BINAL term.

7.5.1.2. BINAL term ::= BINAL factor / BINAL term,and operator,BINAL term.

7.5.1.3. BINAL factor ::= BINAL secondary / not operator, BINAL secondary.

7.5.1.4. BINAL secondary ::= BINAL primary / BINAL primary,shift operator,amount.

7.5.1.5. shift operator ::= left symbol / right symbol.

7.5.2. Semantics

For performing the operations denoted by the or and and symbol applied to binal operands, the binary digits of the sequence which are the values of these operands are ordinally numbered from one upwards,running from right to left. Then the operations are applied to each pair of binary digits which equal ordinal number separately,the or operator yielding zero if the two binary digits are zero and one otherwise,the and operator yielding one if the two binary digits are one, and zero otherwise.The result of the operation is the sequence of the binary digits obtained in this way.For performing the not operation each zero in the sequence is replaced by one and vice versa.

A shift operation is performed as follows:

Step 1: If the value of the amount is a negative integer,then the further elaboration is undefined.

Step 2: The binary digit sequence resulting from the first operand is ordinally numbered from one upwards,running from right to left.

Step 3: The binary digits of the sequence are renumbered by adding to each ordinal number the amount in case of performing the shift denoted by the left symbol,and by subtracting the amount from each ordinal number in the other case.

Step 4: A sequence is constructed whose binary digits are ordinally numbered from one up to the number specified by the reach (cf.3.6.). The positions in that sequence with ordinal numbers equal to those numbers as modified in step 3,are occupied by the corresponding binary digits.The remaining positions are occupied by the binary digit denoted by the zero symbol.This sequence is the result of the indicated shift operation.

7.5.3. Examples

(t binal ∨ h a5b)←3

7.6. Simple alphameric expressions

7.6.1. Syntax

7.6.1.1. simple ALPHAMERIC expression ::= ALPHAMERIC ternary / concatenand,concatenation operator, LONG alphameric ternary.

7.6.1.2. concatenand ::= simple ALPHAMERIC expression.

7.6.1.3. ALPHAMERIC ternary ::= ALPHAMERIC secondary / head operator,ALPHAMERIC secondary.

7.6.1.4. ALPHAMERIC secondary ::= ALPHAMERIC primary / tail operator,ALPHAMERIC secondary.

7.6.1.5. concatenation operator ::= concatenation symbol.

7.6.1.6. head operator ::= head symbol.

7.6.1.7. tail operator ::= tail symbol.

7.6.2. Semantics

The concatenation operation is performed as follows: If one of the operands is the empty string, then the other operand is the result. Otherwise the result is the character sequence which is the value of the first operand followed by the character sequence which is the value of the second operand. The result, however, is undefined if the number of characters contained in this sequence is greater than the number specified by the reach (cf. 3.6.).

Applying the head operator yields a character sequence consisting only of the first character of that sequence which is the value of the operand. If the operand is the empty string, the head operation is undefined.

Applying the tail operator yields a character sequence which is obtained by deleting the first character from that sequence which is the value of the operand. If the latter consists of one character only, the result is the empty string. The application of the tail operator to the empty string is undefined.

7.6.3. Examples

head a alpha conc tail tail a alpha

7.7. Primaries

7.7.1. Syntax

7.7.1.1. SORT primary::= SORT direct denotation/pure SORT primary.

7.7.1.2. pure SORT primary::= changed SORT primary/named SORT primary/ SORT generator/SORT procedure designator/ SORT case expression/ direct SORT primary.

7.7.1.3. changed NONBOOLEAN primary::= lengthened NONBOOLEAN primary/shortened NONBOOLEAN primary/ widened NONBOOLEAN primary/ narrowed NONBOOLEAN primary.

7.7.2. Semantics

For direct denotations which exist for constant array mode and primitive mode only, see 7.8. For changed primaries see 7.17. until 7.20. For named primaries which exist for modes with level at least one see 7.11. For record generators see 7.10. For procedure designators see 7.12. For case expressions see 7.21. For direct primaries see 7.13.

7.8. Direct denotations

7.8.1. Syntax

7.8.1.1. integral direct denotation::= integral value denotation.

7.8.1.2. real direct denotation::= real value denotation.

7.8.1.3. boolean direct denotation::= boolean value denotation.

7.8.1.4. binal direct denotation::= binal value denotation.

7.8.1.5. ALPHAMERIC direct denotation::= alphameric value denotation.

7.8.1.6. long NONCOMPLEX direct denotation::= long symbol, NONCOMPLEX direct denotation.

7.8.1.7. long BINAL direct denotation::= long symbol, BINAL direct denotation.

7.8.1.8. constant array of ORDINARY direct denotation ::= ORDINARY expression list / constant ORDINARY expression list.

7.8.1.9. MODE expression list ::= open symbol, proper MODE expression list, close symbol.

7.8.1.10. proper MODE expression list ::= MODE expression, comma symbol, MODE expression / MODE expression, comma symbol, admissible MODE expression/ admissible MODE expression, comma symbol, MODE expression / proper MODE expression list, comma symbol, MODE expression / admissible MODE expression list, comma symbol, MODE expression / proper MODE expression list, comma symbol, admissible MODE expression.

7.8.1.11. admissible constant CONNECTED expression list ::= proper CONNECTED expression list.

7.8.1.12. admissible RAISED constant ORDINARY expression list ::= proper RAISED ORDINARY expression list.

7.8.2. Semantics

For value denotations see 7.9. For admissible expressions see 5.2. Primitive direct denotations (with the exception of complex direct denotations which do not exist) are given by primitive value denotations possibly preceded by one or more long symbols specifying the length (cf. 3.6.) with which the value should be held in the computer. In case of an alphameric direct denotation, the reach is assumed to be the smallest reach which may comprise the given string, therefore long symbols are omitted.

An array direct denotation is elaborated as follows:

The expressions of the expression list are elaborated in unspecified order. Then, a descriptor is constructed as follows: Let the number of the expressions in the list be u. If the expressions possess values of structure zero, then with the nota-

7.8.2. continued

tions of 6.9.1. the descriptor contains a dimension n = 1, an origin c = 1, and a triple ($l_1 = 1$, $u_1 = u$, $d_1 = 1$).

If the expressions have values with structure greater than zero, then the descriptors associated with these values must be identical in so far as dimensions and lower and upper bounds are concerned. Otherwise the further elaboration is undefined. If this dimension is n' and the lower and upper bounds are l'_j and n'_j, $j = 1,\ldots,n'$, then the new dimension is $n = n' + 1$, the origin $c = 1$ and the new triples are: $l_n = 1$, $u_n = u'_{n-1} - l'_{n-1} + 1$, $d_n = 1$; for $j = n-1, n-2,\ldots,2$: $l_j = 1$, $u_j = u'_{j-1} - l'_{j-1} + 1$, $d_j = u_{j+1} \times d_{j+1}$; and the first triple is: $l_1 = 1$, $u_1 = u$, $d_1 = u_2 \times d_2$.

The ordered set of values resulting from the expression list together with that descriptor is the desired array value.

7.8.3. Extensions

In an expression list, the sequence "integral value denotation, repeat symbol, MODE expression" may be written instead of "MODE expression, comma symbol, MODE expression,..., MODE expression" where all mentioned mode expressions are textually identical and the number of mode expressions in the sequence is identical with the value given by the integral value denotation in the first sequence.

Nevertheless, the mode expression must be elaborated as many times as required by the value given by the integral value denotation.

7.8.4. Examples

```
long long      3.14159265358973238462643
3
3.01
long h   a7o2b5f

begin array  [1:5, 1:5] real a = ( 4 repeat (2,3,3 repeat 0),
                (1, 2, 3, 4, 3.14));
array  [-2:1][-2:1] real b = ((a[1:4][1], a[1:4][2],a[1:4][3],
                (1,2,3,4))) [1:4:-2][1:4:-2]; nil
end
```

7.9. Value denotation

7.9.1. Syntax

7.9.1.1. integral value denotation ::= integral number.

7.9.1.2. real value denotation ::= real number.

7.9.1.3. boolean value denotation ::= truth symbol.

7.9.1.4. binal value denotation ::= binal number.

7.9.1.5. alphameric value denotation ::= string.

7.9.2. Semantics

Value denotations denote values which are considered to exist independently of the program.

The value denoted by an integral value denotation is the integer which in decimal notation is represented by that integral number (for integral numbers cf. 4.1o.1.1.).

7.9.2. continued :

A real number consists of a mantissa followed by an exponent, or of an integral number followed by an exponent, or of a mantissa or an exponent only (cf. 4.10.1.3. up to 4.10.1.6.).

A mantissa is a digit sequence containing a decimal point in front of or within the sequence. The value denoted by a mantissa is the value of the integral number given by the digit sequence, but divided by ten as many times as there are digits following the decimal point.

The value of an exponent is ten raised to the power given by the signed integral number which is part of the exponent.

The value denoted by a real number consisting of an exponent which is preceded by a mantissa or an integral number is the product of the values denoted by the two constituent parts.

The value denoted by a real value denotation is held in the computer with the precision which is specified by the reach derived from the number of long symbols possibly preceding that value denotation (cf. 7.8.). If an integral or real value exceeds the largest value which is possible for that reach (cf. 3.6.), then the value is undefined.

Note that all multiplications, divisions, and exponentiations involved in the determination of a real value must be performed with the precision prescribed by the reach and are to be understood in the sense of numerical analysis.

A boolean value denotation denotes a boolean value, i.e. either true or false.

7.9.2. continued [2]

A binal value denotation denotes a sequence of binary digits each of which is either zero or one. The number of long symbols possibly preceding a binal value denotation determines the reach, i.e. the maximum number of binary digits which may occur in the sequence. If more binary digits occur than specified by the reach, then the binal value is undefined. If the sequence contains a smaller number of binary digits than specified by the reach the lacking number of binary digits is filled up by adding zeros in front of the sequence.

The value denoted by an alphameric value denotation is the character sequence which is part of the string (cf. 4.1.1.7. and 4.1.1.8.). It is recognized that the computer in certain cases adds so-called void characters which have no printing representations at the end of the character sequence. If the string is an empty string, then the denoted alphameric value consists of such void characters only.

When counting the number of characters in a character sequence the void characters are disregarded.

7.1o. Record generators

7.1o.1. Syntax

7.1o.1.1. constant RECORD generator ::= RECORD type identifier, open symbol, RECORD generating part, close symbol.

7.1o.1.2. record with a ORDINARY field generating part ::= ORDINARY initialization.

7.1o.1.3. RECORD and a ORDINARY field generating part ::= RECORD generating part, comma symbol, ORDINARY initialization.

7.1o.2. Semantics

For initializations see 6.3..

The elaboration of a record generator is performed as follows:

Step 1: The expressions forming the generating part are elaborated in unspecified order.

Step 2: A descriptor is constructed as specified by the declaration of that record type given by the record type identifier (cf.6.3.4.) This descriptor contains the field selectors for the created record.

Step 3: The ordered set of values obtained in step 1 is assigned to the ordered set of fields of the new record in unspecified order.

7.1o.3. Extension

If some of the expressions constituting the record generating part are named primaries, then the name symbols in front of these primaries may be omitted.

7.1o.4. Examples

wg21(x real, x real, nil)

7.11. Named primaries

7.11.1. Syntax

7.11.1.1. named reference to a CARRIER primary ::= name symbol, CARRIER identifier / name symbol, CARRIER field selection / name symbol, indexed CARRIER primary.

7.11.1.2. named reference to a procedure with a MODE result and an undetermined number of parameters ::= name symbol,procedure with a MODE result identifier / name symbol,procedure with a MODE result and PARAMETERS identifier.

~~7.11.1.3. named reference to a FREE primary ::= name symbol,FREE primary.~~

7.11.2. Semantics

The value of a named primary is the appellation of that quantity which is identified by the identifier,field selection,indexed primary or procedure identifier.Apart from the declaration,a named primary is the only place where the occurrence of a procedure identifier does not invoke the elaboration of the procedure.

7.11.3. Extensions

For the omission of the name symbol in front of a named primary see 6.3.5., 7.1.3., 7.4.3., 7.1o.3., and 7.12.3..

7.12. Procedure designators

7.12.1. Syntax

7.12.1.1. MODE procedure designator ::= procedure with a MODE result designer / procedure with a MODE result and PARAMETERS designer, open symbol, actual PARAMETERS, close symbol.

7.12.1.2. PROCEDURE designer ::= PROCEDURE identifier / evaluated PROCEDURE primary.

7.12.1.3. actual CONSTANT parameter ::= adjusted CONSTANT expression.

7.12.1.4. actual ORDINARY parameter ::= adjusted reference to a ORDINARY expression.

7.12.1.5. actual PROCEDURE parameter ::= expression symbol, PROCEDURE torso.

7.12.1.6. actual PARAMETERS and PARAMETER ::= actual PARAMETERS, separator, actual PARAMETER.

7.12.2. Semantics

For adjusted expressions see 5.2.. For evaluated procedure primaries see 7.15..

A procedure designator is elaborated in the following steps:

Step 1: If the procedure designer is an evaluated primary, then this primary is elaborated. The procedure which is obtained from that elaboration or which is identified by the procedure identifier is called the "given procedure" and its body (cf. 6.4.) is called the"given body".

Step 2: If there exist neither actual parameters in the procedure designator nor formal parameters in the given body, then step 3 is taken. Otherwise the actual and formal parameters are ordinally numbered from one upwards, and actual and formal parameters with

identical ordinal numbers are called "corresponding parameters". If the numbers of actual and formal parameters are different, then the further elaboration is undefined. Otherwise all actual parameters which are not procedure parameters are elaborated in unspecified order.

Step 3: A copy is made of the given body. This copy is now called the given body. If there exist no formal parameters, then step 8 is taken. Otherwise the copy is modified as specified by step 4 up to step 7.

Step 4: The close symbol at the end of the formal parameters is deleted. Each of the separators between the formal parameters is replaced by the sequence "go on symbol, open symbol". A number of close symbols equal to the number of formal parameters is added at the end of the given body.

Step 5: Each formal declaration is augmented by an equals symbol followed by either an actual parameter, if this is a procedure parameter, or by a denotation of the value of the corresponding parameter as obtained in step 2.

If some of the formal declarations are record declarations containing record indicators, then these record indicators are deleted.

Step 6: If there appears a flexible bound in any of the formal declarators, then the given body is augmented by a close symbol at the end and an open symbol, a declaration for that bound, and a go on symbol at the beginning. The latter declaration is given by the flexible bound followed by an equals symbol and a denotation of that value which corresponds to that flexible bound in the descrip-

tor of the value of the corresponding actual parameter. Then the constant symbol and the integral symbol in the flexible bound are deleted. Step 6 is repeated for each remaining flexible bound.

Step 7: Each declaration in the given body which is obtained by the modifications of step 5 and which does not begin with a constant symbol or a constant procedure symbol is augmented by adding the sequence "constant symbol,reference symbol" in front of it. At each occurrence of an identifier whose defining occurrence (cf. 5.8.2.) is given by a declaration modified in this way , a value symbol is inserted in front of this identifier. If however, the identifier at that occurrence is preceded by a name symbol and is not followed by indices, then this name symbol is deleted and no value symbol is inserted.

Step 8: The given body eventually modified by steps 4 up to 7 must constitute a unitary block. This block is elaborated as if it occurred at the textual position of the procedure designator. If this elaboration is completed, then the resulting value is the value of the procedure designator.In this case and if the declaration of the given procedure has specified the result of the procedure to be an array, a descriptor is constructed from that part of the procedure declaration specifying the mode of the result, and the dimensions and lower and upper bounds contained in this descriptor are compared with the dimensions and lower and upper bounds contained in the descriptor which is associated with the result. If the dimensions or bounds are not identical, the further elaboration is undefined. Otherwise the elaboration of the procedure designator is completed.

7.12.3. Extensions

a) In an actual parameter which is a named primary the name symbol may be omitted.

b) In the place of an actual parameter whose corresponding formal parameter is specified to be a procedure there may also occur a procedure identifier denoting a procedure whose result has the same mode as specified by the corresponding formal declaration.

7.12.3. Examples

For examples see chapter 12.

7.13. Direct primaries

7.13.1. Syntax

7.13.1.1. direct SORT primary ::= nonindexed SORT primary / SORT field selection / indexed SORT primary.

7.13.1.2. direct SIMPLE primary ::= direct constant SIMPLE primary.

7.13.1.3. nonindexed SORT primary ::= SORT identifier / SORT block / evaluated SORT primary.

7.13.2. Semantics

For field selections see 7.14. For evaluated primaries see 7.15. For indexed primaries see 7.16. For blocks see 5.2. and the following sections.

An identifier which stands as a primary is elaborated by taking the value of the identified quantity (see, however, 7.1.2. and 7.11.2.).

7.14. Field selections

7.14.1. Syntax

7.14.1.1. ORDINARY field selection ::= ORDINARY field identifier, of symbol, RECORD primary.

7.14.1.2. constant ORDINARY field selection ::= ORDINARY field identifier, of symbol, constant RECORD primary.

7.14.2. Semantics

A field selection is elaborated as follows:
The constituent record primary is elaborated. If the result is a record which contains within its descriptor a field selector which is denoted by the constituent field identifier,

then the corresponding quantity is selected. Otherwise the further elaboration is undefined. If the field selection stands as a direct primary, then the value of that quantity is taken. Otherwise (cf. 7.1.2. and 7.11.2.) the appellation of that quantity is taken.

7.14.3. Examples

algol x of tony of munich

7.15. Evaluated primaries

7.15.1. Syntax

7.15.1.1. evaluated SORT primary ::= value symbol, nonindexed reference to a SORT primary / value symbol, nonindexed constant reference to a SORT primary / value symbol, reference to a SORT field selection / value symbol, constant reference to a SORT field selection.

7.15.1.2. evaluated procedure with a MODE result and PARAMETERS primary ::= evaluated procedure with a MODE result primary.

7.15.1.3. evaluated procedure with a MODE result primary ::= value symbol, nonindexed reference to a procedure with a MODE result and an undetermined number of parameters primary / value symbol, nonindexed constant reference to a procedure with a MODE result and an undetermined number of parameters primary / value symbol, reference to a procedure with a MODE result and an undetermined number of parameters field selection / value symbol, constant reference to a procedure with a MODE result and an undetermined number of parameters field selection.

7.15.2. Semantics

Evaluation denotes the reduction of the level by one. First the constituent nonindexed primary or the field selection is elaborated. The result of this elaboration must be an appellation. If it is the neutral value, the further elaboration is undefined. If the evaluated primary stands as a reference identification (cf. 7.1.) or as a nonindexed primary, which is followed by indexers, then the mentioned appellation is the result of the evaluated primary. If it stands as a procedure designer (cf. 7.12.), then the result is that procedure to which this appellation belongs. If it stands as a direct primary, then the result is the value to which this appellation refers.

7.15.3. Examples

```
real b;
ref free a = (if k ≤ o then name x else name z);
b ::= val a
```

7.16. Indexed primaries

7.16.1. Syntax

7.16.1.1. indexer ::= subscripter / trimmer.

7.16.1.2. indexed MODE primary ::= subscripted MODE primary / trimmed MODE primary.

7.16.1.3. subscripted MODE primary ::= array of MODE subscribend, subscripter.

7.16.1.4. constant subscripted MODE primary ::= constant array of MODE subscribend, subscripter.

7.16.1.5. trimmed array of MODE primary ::= array of MODE subscribend, trimmer.

7.16.1.6. constant trimmed array of MODE primary ::= constant array of MODE subscribend, trimmer.

7.16.1.7. MODE subscribend ::= nonindexed MODE primary / indexed MODE primary.

7.16.1.8. subscripter ::= sub symbol, subscript, bus symbol.
7.16.1.9. subscript ::= integral expression.
7.16.1.10. trimmer ::= sub symbol, actual lower bound, up to symbol, actual upper bound, at symbol, new lower bound, bus symbol.
7.16.1.11. new lower bound ::= integral expression.

7.16.2. Semantics

Indexing is an operation which yields a quantity or a set of quantities which is a subset of an array. An indexed primary consists of a nonindexed primary followed by a number of indexers. Its elaboration proceeds as follows:

Step 1: The constituent nonindexed primary and all the integral expressions are elaborated in unspecified order. The constituent indexers are consecutively numbered from one upwards.

Step 2: A copy of the descriptor of the array which is given by the nonindexed primary is made. A dimension pointer and an index pointer are both set equal to one, and an index bound is set equal to the number of indexers. If this index bound is greater than the dimension contained in the descriptor, then the further elaboration is undefined.

Step 3: The copy of the descriptor is modified according to the indexer whose ordinal number equals the index pointer, as specified below.

Step 4: The index pointer, and if the indexer last processed is a trimmer, then also the dimension pointer, is increased by one. If the index pointer does not exceed the index bound, step 3 is taken again.

Step 5: If the dimension contained in the modified copy of the descriptor is now positive, the result of the indexing operation is the subset of quantities of the original array which is given by the modified descriptor

7.16.2. continued

together with this descriptor. Otherwise the result is the quantity of the original array whose ordinal number is the origin contained in the copy of the descriptor.

If i is the value of the index pointer and j is the value of the dimension pointer, then the modification mentioned in step 3 is as follows (using the notations of 6.3.3.):
If

the i-th indexer is a subscripter and k is the value of the subscript,

then the further elaboration is undefined, if $k<l_j$ or $u_j<k$. Otherwise:

a) n is decreased by 1
b) c is increased by $(k - l_j)\times d_j$
c) the triple (l_j, u_j, d_j) is removed from the descriptor and the ordinal numbers of the remaining triples, if greater than j, are decreased by 1.

Otherwise, if

the indexer is a trimmer and l, u, l' are the values of the actual lower bound , actual upper bound, and new lower bound respectively,

then the further elaboration is undefined, if $l<l_j$ or $u>u_j$. Otherwise:

a) c is increased by $(l - l_j)\times d_j$
b) l_j is replaced by l'
c) u_j is replaced by $l' + u - l$.

7.16.3. Extensions

a) The sequence "bus symbol, sub symbol" may be replaced by "comma symbol".

b) If in a trimmer the new lower bound is identical with the actual lower bound, then the sequence "at symbol, new lower bound" may be omitted.

7.16.4. Remarks

Note that the indexing operation starts always with a non-indexed primary. Therefore the meaning of the indexed primaries

a [1:n:1] [5]

and

(a [1:n:1]) [5]

is not identical, since (a [1:n:1]) is a nonindexed primary. (If the reader should himself find in a crisis now, Munich 521098 is offered to him).

7.17. Lengthened primaries

7.17.1. Syntax

7.17.1.1. lengthened long NONBOOLEAN primary ::= long symbol, pure NONBOOLEAN primary.

7.17.2. Semantics

The value of a lengthened arithmetic primary is identical with the value of the constituent pure primary, but is now interpreted as having that length which is specified by a number of long symbols one greater than before (cf. 3.7.).

The value of a lengthened binal primary is obtained from the value of the pure primary by adding an appropriate number of zeros in front of the latter value.

7.17.2. continued

The value of a lengthened alphameric primary is obtained from the value of the pure primary by adding an appropriate number of void characters (cf. 7.6.) at the end of the latter value.

7.17.3. Extension

For omitting the long symbol see 7.1.3. and 6.3.5.

7.18. Shortened primaries

7.18.1. Syntax

7.18.1.1. shortened NONBOOLEAN primary ::= short symbol, long NONBOOLEAN primary.

7.18.2. Semantics

The value of a shortened primary is obtained by shortening the value of the primary (cf.3.7.). If the size of the resulting value exceeds the reach given by the prescribed length, then the further elaboration is undefined.

7.19. Narrowed primaries

7.19.1. Syntax

7.19.1.1. narrowed LONG real primary ::= part operator, LONG complex primary.
7.19.1.2. narrowed LONG integral primary ::= round symbol, LONG real primary.
7.19.1.3. narrowed LONG binal primary ::= integral to binal symbol, LONG integral primary.
7.19.1.4. part operator::= real part of symbol / imaginary part of symbol.

7.19.2. Semantics

Narrowing a complex primary means taking the real or imaginary part.

Narrowing a real primary means rounding the value of the real primary to the nearest integral value, i.e. to an integer which differs at most one half from the original real value.

Narrowing an integral primary means representing the value of this primary in the binary number system, i.e. as a sequence of binary digits, which is the resulting binal value. Narrowing an integral value is only defined for nonnegative integral values.

All narrowing operations are only defined if the result does not exceed the reach given by the type and the length of the result.

7.20. Widened primaries

7.20.1. Syntax

7.20.1.1. widened LONG complex primary :: = real to complex symbol, LONG real primary.

7.20.1.2. widened LONG real primary :: = integral to real symbol, LONG integral primary.

7.20.1.3. widened LONG integral primary :: = binal to integral symbol, LONG binal primary.

7.20.2. Semantics

The value of a widened primary is obtained by widening (cf. 3.7.) the value of the constituent primary.

7.20.3. Extensions

a) In the sequence "integral to real symbol, INTEGRAL primary" the "integral to real symbol" may be omitted.
b) In the sequence "real to complex symbol, REAL primary" the "real to complex symbol" may be omitted.

7.21. Case expressions

7.21.1. Syntax

7.21.1.1. MODE case expression :: case symbol, integral expression, of symbol, MODE expression list.

7.21.2. Semantics

For expression list see 7.8.

The elaboration of a case expression is performed in the following steps

Step 1: The integral expression is elaborated. If this elaboration is completed, then step 2 is taken, otherwise step 4 is taken.

Step 2: The constituent expressions of the expression list are ordinally numbered from one upwards. The constituent expression of the expression list whose ordinal number is the value of the integral expression, is selected and step 3 is taken. If no expression with this ordinal number exists in the expression list, then the further elaboration is undefined.

Step 3: The selected expression is elaborated. If this elaboration is completed, then the value of this expression

7.21.2. continued

is the value of the case expression and the elaboration of the case expression is completed. Otherwise step 4 is taken.

Step 4: The elaboration of the case expression is terminated.

7.21.3. Example

```
a := x + case 1 of (1,2,3,4)
```

8. Representations

In this section the representations of some basic symbols of the language are given. The first column contains the denotation of the symbols used in definning the strict language, the other columns contain one or more possible representations of these symbols.

8.1. Listing of representations

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

8.1. continued 1

zero symbol	o		
one symbol	1		
two symbol	2		
three symbol	3		
four symbol	4		
five symbol	5		
six symbol	6		
seven symbol	7		
eight symbol	8		
nine symbol	9		
ten symbol	a		
eleven symbol	b		
twelve symbol	c		
thirteen symbol	d		
fourteen symbol	e		
fifteen symbol	f		
true symbol	true		
false symbol	false		
dyadic plus symbol	+		
dyadic minus symbol	-		
times symbol	×		×
divided by symbol	/		
integral divided by symbol	÷		div
modulo symbol	mod		
to the power symbol	↑	**	power
times ten to the power symbol	$_{10}$	↟	ten
point symbol	.		
absolution symbol	abs		
plus i times symbol	i		
monadic plus symbol	+		
monadic minus symbol	-		
equals symbol	=		equal
differs symbol	≠		not equal
conforms symbol	::		conform
not conforms symbol	:/:		not conform

8.1 continued 2

conforms and becomes symbol	::=	
becomes symbol	:=	
less symbol	<	less
not less symbol	≮	not less
greater symbol	>	greater
not greater symbol	≯	not greater
not symbol	¬	not
and symbol	∧	and
or symbol	∨	or
left symbol	←	left
right symbol	→	right
concatenation symbol	conc	
tail symbol	tail	
head symbol	head	
long symbol	long	
short symbol	short	
value symbol	val	
name symbol	name	
binal to integral symbol	bti	
integral to real symbol	itr	
real to complex symbol	rtc	
real part of symbol	re	
imaginary part of symbol	im	
round symbol	round	
integral to binal symbol	itb	
array symbol	array	
free symbol	free	
constant symbol	const	
constant procedure symbol	proc	
expression symbol	expr	
reference symbol	ref	
record type symbol	rec type	
record symbol	rec	
integral declaration symbol	int	
real declaration symbol	real	

8.1. continued 3

complex declaration symbol	compl	
binal declaration symbol	bits	
binary symbol	b	
octal symbol	o	
hexadecadic symbol	h	
alphameric declaration symbol	alpha	
boolean declaration symbol	bool	
neutral symbol	nil	
open symbol	(	begin
close symbol	)	end
if symbol	if	
then symbol	then	
else symbol	else	
case symbol	case	
of symbol	of	
parallel open symbol	parbeg	
parallel close symbol	parend	
parallel separation symbol	:	
sub symbol	[	
up to symbol	:	
at symbol	:	
bus symbol	]	
label symbol	:	
comma symbol	,	
go on symbol	;	
completion symbol	break	fin
go to symbol	go to	
quote symbol	'	
comment begin symbol	#(	(/
comment end symbol	)#	/)
for symbol	for	
from symbol	from	
by symbol	by	
to symbol	to	
do symbol	do	

8.1. continued 4

fi symbol	fi
while symbol	while
except symbol	exc
repeat symbol	repeat

8.2. Remarks

The fact that the representations of the letters, given above, are usually referred to as small letters is not meant to imply that the so-called corresponding capital letters could not serve equally well as representations.
On the other hand, if both a smaller letter and the corresponding capital letter occur, then one of them is considered to be the representation of an "other letter"(cf.4.2.).
If in a text intended to be a program a character occurs outside a string or a comment which does not match one of the given symbol representations and is also not a typographically display character such as blank space, change to a new line, and change to a new page, then it is to be interpreted as an other letter.
If a letter is not syntactically admissible in that position, then the text is not a program.
A representation which is a sequence of underlined or boldfaced characters is different from the sequence of those characters when not underlined or in boldface.

9. Further extensions

(a) If e denotes an expression of any sort in which the identifiers i,k and rep do not occur, if further a, b, c and d denote integral expressions and l denotes a boolean expression, then the following replacements may occur:

(i)
```
begin const array [1:3] int i = (a,b,c); int k;
        k := i[1];
rep: if (k - i[3]) × i[2] ≤ 0
     then if l
          then begin const int j = k;
                     e; k := k + i[2]; go to rep
               end
          fi
     fi;
end
```
may be replaced by
```
for j from a by b to c while l do e
```

(ii)
```
begin const array [1:2] int i = (a,b); int k;
        k := i[1];
rep: if l then
     begin const int j = k;
        e; k := k + i[2]; go to rep
     end
     fi;
end
```
may be replaced by
```
for j from a by b while l do e
```

(iii)
```
begin const array [1:4] int i = (a,b,c,d); int k;
    k := i[1];
rep: if (k - i[3]) . i[2] ≤ o
    then begin const int j = k;
            if j ≠ i[4] then e fi;
            k := k + i[2]; go to rep
        end
    fi;
end
```

may be replaced by

```
for j from a by b to c exc d do e
```

(b) If one of the extensions mentioned above is used then

(i) "for j" may be omitted if the identifier j is not used in e;

(ii) "from 1" may be omitted;

(iii) "by 1" may be omitted.

(c) If the extension (a)(i) is used, then "while true" may be omitted.

10. Environment enquiries

(This chapter was not worked out)

11. Input/output

(This chapter was not worked out)

12. Examples

1)
```
proc real prod(const array[1:int n] real a,b);
#( the inner product of two vectors a and b with n
   elements )#
begin long real x = 0;
for k to n do x := x + long ( a[k] ) × long ( b[k] );
short x
end;

#( a procedure call of prod with :
array[1:5] real a1 = ( 1,2,3,4,5) and
array[1:5] real a2 = ( 5,4,3,2,1) is :  prod(a1,a2) )#
```

2)
```
proc array[1:n] int order( array[1:n] int a)
#( the elements of a vector a of length n are
   ordered corresponding to the absolute value
   of these elements where n is a global parameter )#
begin int u,v,y = n;
repeat : v := abs a[y];
for i to y - 1 do if v≥ abs a[i] then
( u := v; v := abs a[i]; a[i] := u ) fi;
a[y] := v;
y := y -1; if y ≥ 1 then go to repeat else ^
end;

#( a procedure call of order with :  int n = 10;
array[1:10] int a = ( 10,1,0,2,3,5,8,1,5,7) is: order( a ) )#
```

12 continued 1

3) proc real spur(array[1:int n,*] real a);
(real s = 0; for k to n do s := s + a[k,k]; s);

4) proc transpose(array[1:int n] real a);
begin real w = 0;
for i to n do
for k := 1 + i to n do
(w := a[i,k];
a[i,k] := a[k,i];
a[k,i] := w)
end;

5) proc euler(const int tim, const real eps, real sum) function(proc real fct);
begin int i = 0 , n = 0 , t = 0 , k; real mn , mp , ds;
array[0:15] real m;
m[0] := fct(0); sum := m[0]/2;
nextterm: i := i + 1; mn := fct(i);
for k from 0 to n do
(mp := (mn + m[k])/2; m[k] := mn);
if (abs mn < abs m[n])∧(n < 15) then
(ds := mn/2; n := n + 1; m[n] := mn) else ds := mn;
sum := sum + ds; if abs ds < eps then t := t + 1 else
t := 0; if t < tim then goto nextterm fi
end;

```
6)  begin #( frame of a program concerning symbolic differentiation )#
        rec type (real value) constant;
        rec type ( ref free left operand, int operator,
                   ref free right operand) triple;
        rec type (alpha bound variable, ref free body) function;
        rec type (rec (function) function name,
                  ref free parameter) function designator;
    rec type (rec (triple)t ) loctriple; rec (loctriple)tr;
    const int plus = 1, minus = 2, times = 3, divided by = 4;
        rec (constant) zero = constant(0), one = constant(1);
        proc ref free derivative(const ref free e)with re
                                        spect to(const alpha x);
        begin rec (constant) ec; alpha ev; rec (triple) et;
              rec (function designator) ef;
              if ec :: val e then name zero else
              if ev :== val e then if ev = x then name one else
                        name zero else
              if et :== val e then
              begin ref free u = left operand of et;
                    ref free v = right operand of et;
                    ref free udash = derivative(u,x);
                    ref free vdash = derivative(v,x);
  tr := loctriple (case operator of et of
                    (triple(udash, plus, vdash), triple(udash, minus,
                    vdash), triple(triple(u, times, vdash), plus,
                    triple(udash, times, v), triple(triple(udash,
                    divided by, v), minus, triple(triple(vdash, times,
                    u), divided by, triple(v, times, v))))); name t of tr
              end else
              if ef :== val e then
```

```
            begin rec (function) f = function name of ef;
                  ref free g = parameter of ef;
                  rec (function) fdash = function(bound variable of f,
                  derivative(body of f, bound variable of f));
tr := loctriple (triple(function designator(fdash, g), times,
                  derivative(g, x))); name t of tr
            end fi
      end # ( end of declarations concerning differentiation ) #
end
```

11. 5. 67

Errata 1 to "Another Proposal for ALGOL 67"

by G. Goos et al.

page	line	
17		replace "an" by "a" in metaproduction 2.1.34. (twice)
20	11 f.b.	read "certain" instead of "certaion"
29		delete "string/" from production 4.1.1.1.
38	12 f.b.	read "is" instead of "es"
40		replace the production form 5.3.1.10 by "statement :: proper SORT expression/ conditional statement/ jump."
41	5 f.a.	insert "than" between "entity" and "it"
44	6/7 f.a.	add the new production forms "5.5.1.3. conditional statement :: if clause, expression, else symbol, expression. 5.5.1.4. expression:: SORT expression."
56	6 f.a.	insert "," between "symbol" and "PROCEDURE"
56		insert "adjusted" between "::" and "MODE" in production form 6.4.1.5.
56		replace "admissible" by "adjusted" in production form 6.4.1.6.
57	6 f.a.	replace "symbol" by "torso"
58	6 f.a.	replace "within" by "with"
76	2 f.b.	read "sequence"instead of "sequenc"
78		delete production form 7.11.1.3.
87	1 f.b.	replace "1" by "l"
98	2/3 f.a.	read "integral expressions" instead of "inte-expression"
98	11 f.a.	read "go to" instead of "go to"
98	6 f.b.	read "go to" instead of "go to"
98/99		insert a";" between "fi" and "end" in all three extensions
101	3/9/13/23	replace "(" by "#("
101	4/11/15/24	replace the last ")" by ")#"
102	1 f.a.	replace "1: int n,n" by "1: int n,:"
102	2 f.a.	insert "for k" between ";" and "to n"
102	7/8/15 (12/) f.a.	add ";" at the end

page	line		
102	11	f.a.	read "tim" instead of "tim"
102	5	f.b.	insert "^" between ")" and "("
103	7/8	f.a.	add "rec type (rec (triple)t) loctriple; rec (loctriple)tr;"
103	8	f.b.	add in front of "case" the sequence "tr:= loctriple("
103	3	f.b.	add at the end of the line "); name t of tr"
104	5.	f.a.	add in front of "triple" the sequence "tr:= loctriple ("
104	4	f.b.	add at the end of the line "); name t of tr"

Add a "↑" in front of the production forms 4.1.1.2. (page 29) and 5.2.1.1. (page 37).