MATHEMATICAL CENTRE TRACTS

16

MATHEMATICAL CENTRE TRACTS

16

FORMAL DEFINITION OF PROGRAMMING LANGUAGES

WITH AN APPLICATION TO THE DEFINITION OF ALGOL 60

BY

J. W. DE BAKKER

2nd Edition

MATHEMATISCH CENTRUM AMSTERDAM

1970

First printing in 1967

Acknowledgements

I wish to express my gratitude to the Mathematical Centre for the opportunity of performing the research dealt with in this paper.

I am deeply indebted to Prof. Dr. Ir. A. van Wijngaarden, whose work formed the basis for these investigations and from whose many invaluable suggestions and criticisms I have greatly profited. To B.J. Mailloux I am grateful for many helpful discussions and for the correction of the English text.

I also would like to thank Mrs. H. Roqué for the excellent typing of the manuscript and Mr. D. Zwarst for the printing.

Amsterdam, 1967 J.W. de B.

Preface to the second printing

This is an unaltered edition of the first printing. The discussion of other methods for formal language definition, as contained in chapter 1, has been considerably extended in:

J.W. de Bakker, Semantics of Programming Languages, in Advances in Information Systems Science, vol. 2 (J.T. Tou, ed.), pp. 173-227, Plenum Press, New York, 1969.

Amsterdam, 1970 J.W. de B.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

In this paper we treat a method for the formal definition of syntax and semantics of programming languages. As an important application, a complete formal definition of ALGOL 60 is given.

We can distinguish between two aspects of the formalization of a programming language:

1. Formalization of the syntax: Given an alphabet, i.e. a finite set of symbols, to exhibit a set of rules which define which sequences of symbols over this alphabet constitute a program in the language concerned.
2. Formalization of the semantics, i.e. introduction of a formal system which defines the meaning of a (syntactically correct) program.

A fairly satisfactory solution to the first problem was given in the ALGOL 60 report [38], in which the syntax of ALGOL 60 was defined by means of a formalism due to Backus [2].
The notation of Backus has been used subsequently for the definition of several other programming languages and also for the syntactical definition of related formal systems. As was proved later [23], Backus notation is equivalent to a concept which had been introduced previously by Chomsky [11], viz. that of context free grammar, which is a specialization of the notion of phrase structure grammar, also due to Chomsky [10]. However, Backus notation is not entirely sufficient for the definition of the syntax of programming languages, such as ALGOL 60. In fact, the ALGOL 60 report contains, besides the rules formalized in Backus notation, several others, expressed in English, which impose further restrictions on the class of syntactically correct programs. It can be proved that it

is impossible to include these further restrictions in a context free grammar [20].
A generalization of the notion of context free grammar, which still needs some rules stated in English, but which allows considerably more formalization of the syntax, has been given by van Wijngaarden [50].
Another extension of context free grammars, which also makes it possible to formalize rules which cannot be expressed in Backus notation, has been proposed by Caracciolo di Forino [6].

After the problem of the formalization of the syntax had been (partly) solved, it seemed natural to try and find a formalism for the definition of the semantics of programming languages. In the ALGOL 60 report, the semantics is described entirely in English. However, there exists a fairly general agreement that this is unsatisfactory and that it is desirable to formalize the semantics (maybe only a part of it) as well.In fact, it soon appeared that the description in English shows several defects, mainly apparent from the fact that various constructions may be thought of, for which the ALGOL 60 report does not give an unambiguous interpretation. A list of these ambiguities (which is not even complete) has been given by Knuth [27].
In the past few years, several systems for the formalization of the semantics of programming languages have been proposed. However, there exists no agreement at all on what one means by a semantical description of a programming language. In September 1964, a conference on "Formal Language Description Languages" was held, organized by the technical committee on programming languages of the International Federation for Information Processing. The proceedings of this conference [41] show clearly how much the ideas of the several authors diverge.
Landin [30, 31, 32], Böhm [4, 5] and Strachey [43] use the λ-calculus of Church as the basis of their formalisms. Essentially, this means that they try to describe a program by means of a functional notation. However, in our opinion this conflicts with the dynamic structure of a program, which consists of a number of instructions executed successively. (This criticism has also been given by Wirth [47].) In defence of the use of the

λ-calculus it should be mentioned that it can be used to describe the locality concept of ALGOL 60 in a way which is more elegant than in any other system of which we know. Assignment statements and goto statements on the other hand can be included in the system only with considerable difficulty. We may add to this that the paper of Landin [32] forms the most completely worked out system for the definition of ALGOL 60 which has been proposed up to now.

Steel [40, 42] has given the foundations for his way of formalizing semantics, without, however, showing how fundamental concepts in programming languages can be described with his system.

McCarthy [34, 35] also gives only simple examples, from which it is difficult to conclude whether his mechanism is sufficient for the more complicated concepts of a programming language such as ALGOL 60, e.g. the meaning of declaration, of recursive procedures, or the call by name concept. McCarthy introduces the notion of a state vector, the components of which are: the current values of the variables which occur in the program, and the number of the statement which is to be executed. He admits, however, that the above mentioned concepts will require a more complicated state vector [1)].

Wirth [47] lets the semantical description of a programming language run parellel to its syntactical definition. Whenever a syntactical rule is applied during the analysis of a program, a corresponding semantical rule is applied which changes the values of zero or more entities in a so-called environment. The semantical rules are formalized in a language which is said to correspond closely to the elementary operations of a computer. It is assumed that the concepts of this elementary language do not need further formal definition. As possible objections to his approach we might mention: As Wirth himself admits, it is applicable only to programming languages whose syntax is less general than that of a context free grammar. Also, it appears that the system is not entirely sufficient for the treatment of the main example he gives, namely of the language EULER, a generalization of ALGOL 60 (see also [46]). First, he has to extend his elementary notation with a number of operators and types, the

---

1) A combination of the formalisms of Landin and McCarthy has been used for the formal definition of PL/I, see [53].

meaning of which is in our opinion not so obvious that they belong in this elementary system. Furthermore, the definition of the meaning of an EULER program is given in two phases, the second of which does not have the structure as described above, since in this phase the semantical rules are not applied in parallel to the syntactical ones.
When we compare Wirth's method to the system we propose in this paper, it appears that his degree of formalization is considerably less than ours. Concepts which he considers too elementary to need further formal definition, have been treated formally in our system. On the other hand, his mechanism is of greater practical importance, since a definition of a programming language with his method can be used as the basis for a compiler for that language. We shall see that it is not at all easy to do the same with our system.
Analogous considerations hold for the work of Feldman [19]. The "Formal Semantic Language" which he uses to define the semantics of programming languages, has been designed for the purpose of constructing compilers. For these practical problems, FSL has proven to be of much use. However, we feel that FSL is too complicated a language to be considered a solution to the problem of the formalization of semantics.
Garwick [21] wants to define programming languages by means of their compilers, which are supposed to be machine independent. An abstract computer must be introduced, the code of which is used to write this compiler. The output of the compiler must also be in this code. However, he omits all details of the properties of this code. Moreover, it is doubtful whether the concept of translation, however great its importance be in practice, should be used for the definition of a formal language.
Nivat and Nolin [39] define the semantics of ALGOL 60 in several steps. First an ALGOL 60 program is translated into a program written in so-called ALGOL ε. The result is translated into an ALGOL η program. ALGOL η resembles an assembly language so much that further definition is unnecessary.
Finally, we mention some investigations of a more theoretical nature which have been inspired by problems concerning the semantics of pro-

gramming languages: Elgot [17], Elgot and Robinson [18], Igarashi [25, 26], Thiele [44] and Yanov [51].

The system that we treat in this paper is based on two papers by van Wijngaarden [48, 49]. We quote from [48]:
"The definition of a language should be the description of an automatism, a set of axioms, a machine or whatever one likes to call it, that reads and interprets a text or program, any text for that matter, i.e. produces during the reading another text, called the value of the text so far read. This value is a text that changes continuously during the process of reading and intermediate stages are just as important to know as the final value".
This idea is worked out as follows [1)]:
An abstract machine is introduced, which in the sequel will be called "processor". A text which is offered to the processor for evaluation is called a "name". A number of symbols have the property that their occurrence in a name causes a special reaction of the processor. Such a special symbol is e.g. the so-called metacomma, denoted by <u>co</u>. A name will consist in general of a sequence of so-called simple names, which are separated by these metacommas. The evaluation of a name is performed by successive evaluation of the simple names which constitute it. The value of a simple name is determined by consulting a list of rules, the so-called "list of truths", which list will be called V in the sequel. This list, V, is initially empty and is filled during the evaluation of a name with the values of the simple names which constitute it. The way in which V is consulted to determine the value of a simple name may provisionally be summarized as follows: The list of truths has essentially the same structure as a Markov algorithm [33], i.e. it is a list of rules, consisting of a left and right part, separated by the symbol <u>is</u>. These rules are applied in the same way as with a Markov algorithm. However, an important extension has been introduced, namely the possibility of using metalinguistic variables (in the sense of Backus) in these left

---

1) The following description is intended to give only a first impression of the system. Precise definitions will be given in the next chapters.

and right parts [1]. Moreover, the definition of the values which may be assumed by these metalinguistic variables is done by means of rules which also form part of V. Another new feature is the possibility of having the application of a truth in V depend on a condition, also belonging to this truth.

The formalism, of which we have sketched some principles above, may itself be considered as a formal language. Since this language is used for the definition of other languages, we shall call it in the sequel "the metalanguage".
A complete description of the metalanguage is given in chapter 2. Comparison with [48, 49] will show that some changes have been introduced. First of all, the idea of a preprocessor has been done away with. This was used in [48, 49] to reduce, by means of a non-formalized process, concepts which are logically redundant, to more fundamental ones. Since we wish to give a complete formal definition of ALGOL 60, we cannot use the preprocessor. Also, there no longer appear any "loose remarks concerning locality and so on", which were supposedly present in V in [49]. Some changes in the notation have been adopted, to avoid confusion between symbols in the language which is to be described (e.g. the symbols "=" and "," in ALGOL 60) and symbols in the metalanguage (e.g. <u>is</u> and <u>co</u>). Furthermore, we have defined the meaning of a condition in a truth somewhat more precisely than in [48] or [49]. Concerning this definition it should be remarked that it is certainly the least elegant concept of the metalanguage. However, we use it extensively in the definition of ALGOL 60 and we have not succeeded in replacing it by another one which fits better with the other concepts.
Chapter 2 starts in section 1 with a description of the syntax of the metalanguage, by means of a context free grammar. Section 2 gives some syntactical examples. In section 3 the semantics of the metalanguage is described

---

1) A combination of Markov algorithms and context free grammars has been proposed subsequently also by Caracciolo di Forino [7, 8, 9] and Cohen and Wegstein [15]. Similar concepts occur in the language AMBIT [14]. The first application of Markov algorithms to programming seems to be due to Yngve, in his design of Comit [52].

in English. Section 4 contains some simple examples, such as the definition of the Euclidean algorithm for the greatest common divisor of two natural numbers.

In chapter 3 we derive some properties of the metalanguage. In section 1 we consider three definitions of effective computability, i.e. Markov algorithms, Turing machines and recursive functions. We prove for each of these concepts that it can be defined by means of the metalanguage. We do not treat the reverse problem, i.e. we have not investigated whether it is possible to define the metalanguage in terms of one of these three systems.
In section 2 we consider the relation between the metalanguage and a few concepts of the theory of phrase structure grammars. From a theorem of Chomsky, namely that each phrase structure language is a recursively enumerable set [11], and the results of section 1, it follows directly that each phrase structure language can be defined by means of the metalanguage. The relation between context free grammars and a concept from the metalanguage is then studied in more detail. An example is given of the use of the metalanguage for the definition of a context sensitive grammar. The classification of Chomsky of phrase structure grammars in four types and their defining abstract machines are introduced. Each of these abstract machines is defined in terms of the metalanguage.

The formal definition of the metalanguage follows in chapter 4, section 1. There the processor is defined by means of an ALGOL 60 program which acts both as a definition and as an implementation of the metalanguage. The description in English of the metalanguage in chapter 2 should therefore not be considered to be its definition proper. Thus, the metalanguage is defined on the one hand by an ALGOL 60 program, and on the other hand it is used (in chapter 5) to define ALGOL 60.
One might imagine the following picture of this situation: We introduce a "language space", the elements of which are the possible interpretations of ALGOL 60. Suppose one wants to use our system to learn the semantics of ALGOL 60. We assume that he has a provisional knowledge of it, based on the ALGOL 60 report. This means that he finds himself in a certain

point in the language space, say P_0. With this knowledge he can understand the working of the processor, and hence also the definition of ALGOL 60 in chapter 5. After the study of this definition, he will have obtained a new idea of the semantics of ALGOL 60, i.e., he finds himself now in a point P_1. Next he again reads the program for the processor, and chapter 5, after which he will have reached a point P_2, etc. Suppose one finds oneself after i steps in point P_i ($i \geq 1$). We distinguish three cases:

1. $P_i = P_{i-1}$. This means that $P_n = P_{i-1}$, for $n \geq i$.
   The process converges, i.e. it yields a fixed interpretation of ALGOL 60. Generally, this fixed point P_i will depend upon the initial point P_0.
2. $P_i = P_{i-k}$, $k > 1$, $P_i \neq P_{i-1}$.
   The process diverges; it is not possible to obtain a fixed interpretation of ALGOL 60.
3. Neither 1, nor 2 occurs. No decision can be taken, and a next step has to be performed.

It is not possible to describe this iteration process more formally. This is caused by the fact that the ALGOL 60 program which defines the processor contains input/output operations which are not treated in the formal definition of ALGOL 60 in chapter 5, since they do not form part of it.

In section 2 of chapter 4 the working of the processor is demonstrated by several examples. Some of these examples have been discussed already in chapters 2 and 3. Also, some very simple parts from the definition of ALGOL 60 in chapter 5 are treated. Both time and space restrictions of present day computers prohibit the running of larger parts of the ALGOL 60 definition, let alone the whole of it.

Chapter 5 gives the complete formal definition of ALGOL 60; explanations of this definition follow in chapter 6. In chapter 6 we first treat some shortcomings of the definition. Then in sections 2 to 6 we give a general survey of its structure. The remaining sections of chapter 6 comment upon each of the sections of chapter 5. The main difficulties in the definition of ALGOL 60 proved to be: the locality concept and the goto statements. Assignment statements, on the other hand, fit in very naturally with the metalanguage.

A judgment on the merits of the metalanguage as a means of describing the semantics of programming languages will depend on the requirements which one imposes upon such a description. If one wants a mechanism from which a compiler for the language concerned can easily be derived, then our system is certainly not the solution. The value of the metalanguage consists in its ability to give a complete and precise definition of the whole language, containing all concepts, from the addition and subtraction of integers to the treatment of procedures. Such a complete definition will always be rather large. It should be added here that several aspects of the semantics of ALGOL 60, which are of no essential importance, have complicated and lengthened the definition in chapter 5 considerably. If a programming language were designed with the metalanguage as the presupposed tool for semantic description, then such a description could be substantially shorter.
Recently, suggestions have been made for the introduction of programming languages which allow the programmer to include modifications or extensions of the language in his program. Such an interaction between language and program may also be described very well by the metalanguage.

CHAPTER 2

DESCRIPTION OF THE METALANGUAGE

In section 1 of this chapter we define the syntax of the metalanguage. In section 2 we give some syntactical examples. Section 3 describes the semantics of the metalanguage, some concepts of which are explained by means of a few simple examples in section 4.

2.1. Syntax of the metalanguage

The syntax of the metalanguage is defined by means of a context free grammar, written in Backus notation.

```
1. <NAME>::= <SIMPLE NAME>|<SIMPLE NAME> co <NAME>

2. <SIMPLE NAME>::= tr|<METASTRING>|<SIMPLE TERM>

3. <METASTRING>::= {<LIST OF METAEXPRESSIONS>}

4. <LIST OF METAEXPRESSIONS>::= <METAEXPRESSION>|
                                <METAEXPRESSION>co<LIST OF METAEXPRESSIONS>

5. <SIMPLE TERM>::= <SIMPLE FACTOR>|<SIMPLE FACTOR>in<SIMPLE METAVARIABLE>

6. <SIMPLE FACTOR>::= <TERMINAL SYMBOL>|va{<TERMINAL SEQUENCE>}|
                      <TERMINAL SYMBOL><SIMPLE FACTOR>|
                      va{<TERMINAL SEQUENCE>}<SIMPLE FACTOR>

7. <METAEXPRESSION>::= <CONDITION> im <LEFT PART> is <RIGHT PART>|
                       <CONDITION> im <LEFT PART>|
                                      <LEFT PART> is <RIGHT PART>|
                                      <LEFT PART>

8. <CONDITION>::= tr|<METASEQUENCE>
```

9. <LEFT PART>::= <METASEQUENCE>|<METASEQUENCE> in
<SIMPLE METAVARIABLE>

10. <RIGHT PART>::= <SIMPLE RIGHT PART>|{<LIST OF SIMPLE RIGHT PARTS>}

11. <LIST OF SIMPLE RIGHT PARTS>::= <SIMPLE RIGHT PART>|
<SIMPLE RIGHT PART> co <LIST OF SIMPLE RIGHT PARTS>

12. <SIMPLE RIGHT PART>::= tr|<METASTRING>|<INDEXED METATERM>

13. <INDEXED METATERM>::= <INDEXED METAFACTOR>|
<INDEXED METAFACTOR> in <SIMPLE METAVARIABLE>

14. <INDEXED METAFACTOR>::= <INDEXED METASEQUENCE>|
va{<INDEXED METASEQUENCE>}|
<INDEXED METASEQUENCE><INDEXED METAFACTOR>|
va{<INDEXED METASEQUENCE>}<INDEXED METAFACTOR>

15. <TERMINAL SEQUENCE>::= <TERMINAL SYMBOL>|
<TERMINAL SYMBOL><TERMINAL SEQUENCE>

16. <METASEQUENCE>::= <TERMINAL SYMBOL>|<METAVARIABLE>|
<TERMINAL SYMBOL><METASEQUENCE>|
<METAVARIABLE><METASEQUENCE>

17. <INDEXED METASEQUENCE>::= <TERMINAL SYMBOL>|<INDEXED METAVARIABLE>|
<TERMINAL SYMBOL><INDEXED METASEQUENCE>|
<INDEXED METAVARIABLE><INDEXED METASEQUENCE>

18. <METAVARIABLE>::= <NON INDEXED METAVARIABLE>|<INDEXED METAVARIABLE>

19. <NON INDEXED METAVARIABLE>::= <SIMPLE METAVARIABLE>|
<OPTIONAL METAVARIABLE>

20. <INDEXED METAVARIABLE>::= <INDEXED SIMPLE METAVARIABLE>|
<INDEXED OPTIONAL METAVARIABLE>

21. <TRUTH>::= tr|<METAEXPRESSION>

22. <LIST OF TRUTHS>::= <TRUTH>|<TRUTH> co <LIST OF TRUTHS>

23. <DERIVED CONDITION>::= tr|<METASEQUENCE>

24. <DERIVED SIMPLE RIGHT PART>::= <SIMPLE NAME>|<EMPTY>

25. <DERIVED RIGHT PART>::= <DERIVED SIMPLE RIGHT PART> | {<NAME>}

26. <SIMPLE PRIMARY>::= va{<TERMINAL SEQUENCE>}

27. <SIMPLE SEQUENCE>::= <TERMINAL SEQUENCE> |
<TERMINAL SEQUENCE> in <SIMPLE METAVARIABLE>

For the denotation of the syntactic entities in this grammar we have used sequences of (capital) metametaletters, enclosed between the metametabrackets "<" and ">". Whenever we use these sequences in the sequel they will refer to the corresponding syntactic definitions. It is understood that the use of the English language may lead to deviations from these words; for example, the use of lower case letters or of plural forms.

The entities in the left hand sides of 21 to 27 are introduced for reference purposes only.

<EMPTY> denotes the empty sequence.

A simple or optional metavariable is denoted by a sequence of metaletters, enclosed between the metabrackets "<" and ">", or "≤" and "≥" respectively.

An indexed simple metavariable or an indexed optional metavariable is denoted by a sequence of metaletters, followed by a sequence of metadigits, the whole enclosed between the metabrackets "<" and ">", or "≤" and "≥" respectively.

The set of terminal symbols is given in chapter 4. Essentially, one may choose for this set any finite, non empty set of symbols which is disjoint from the set of metaconstituents (see below).

However, in chapter 4 we define the set that is accepted by the ALGOL 60 program that defines the processor.

The set of metaconstituents consists of:

a. The metasymbols im, in, is, va, co, tr, {, }, ⦃, ⦄.

b. The metavariables.

We introduce the following terminology which is used in the next sections:

a. Small Greek letters stand for syntactic entities (i.e. metametalinguistic variables), capital Roman letters for metavariables or terminal symbols, and small Roman letters for terminal symbols.

b. For any metavariable A, $\bar{A}$ is the non indexed metavariable which results from A by deleting the metadigits, if any, in the denotation of A.

For any (indexed) optional metavariable A, $\tilde{A}$ is the (indexed) simple metavariable which results from A by replacing in its denotation the metabrackets "$\underline{<}$" and "$\underline{>}$" by "<" and ">".

Example: Let A be the indexed optional metavariable $\underline{<}$identifier1$\underline{>}$.
Then $\bar{A}$ is $\underline{<}$identifier$\underline{>}$, $\tilde{A}$ is <identifier1> and $\tilde{\bar{A}}$ is <identifier>.

c. Two indexed metavariables are called similar, if their denotations differ at most in the enclosing metabrackets.

Example: $\underline{<}$expression2$\underline{>}$ and <expression2> are similar, but <expression1> and <expression2> are not similar.

d. A simple sequence can have the form <TERMINAL SEQUENCE> or <TERMINAL SEQUENCE> in <SIMPLE METAVARIABLE>. In both cases we call the terminal sequence "the terminal sequence of the simple sequence". In the second case we call the simple metavariable "the simple metavariable of the simple sequence". Similarly, we define the metasequence and the simple metavariable of a left part.

Example: The terminal sequence of "abc in <identifier>" is "abc", and its simple metavariable is "<identifier>".
The metasequence of "<primary> in <factor>" is "<primary>" and its simple metavariable is "<factor>".

2.2. Syntactical examples

In this section we use the set of terminal symbols given in chapter 4.

Name:

{1 + 1 is 2 co 2 + 1 is 3} co 1 + 1 co 2 + 1 co {3 + 1 is 4}

Simple name:

tr

{2 - 1 is 1}

2 - 1

Metastring:

{1 + 1 is 2 co 2 + 1 is 3}

Simple term:

va{4 + 3} - va{3 + 4}

abc in <identifier>

Simple factor:

- va{a + b}

3 + 2

Metaexpression:

<letter> in <identifier>

a is ⦃ b is ⦃ c is d ⦄⦄

<tape1> 1 : 1 <tape2> is <tape1> a 1 <tape2>

<state1><symbol1><symbol2><state2> im
<tape1><state1><symbol1><tape2> is
<tape1><state2><symbol2><tape2>

<letter1> pre <letter2> im
<letter1><word> pre <letter2><word>
(note that the underlined symbol pre is not a metasymbol, but a terminal symbol)

Condition:

<letter1> pre <letter2>

tr

Left part:

<letter1><word> pre <letter2><word>

<block> in <program>

Right part:

a

<tape1><state2><symbol2><tape2>

{R co S co T}

{⦃a is ⦃ b is {⦃c is {d co e}⦄ co f}⦄⦄ co g}

Indexed metaterm:

<letter1> in <identifier>

Indexed metafactor:

va{<digit1><pm1>1}<pm1> va{<digit2>-1}

Terminal sequence:

a + bc - d

Metasequence:

<letter1><word>

Indexed metasequence:

<state1><symbol1> L <state2>

Simple metavariable:

<identifier>

Optional metavariable:

<identifier>

Indexed simple metavariable:

<identifier21>

Indexed optional metavariable:

<identifier8>

2.3. Semantics of the metalanguage

We introduce an abstract machine, called the processor, which is defined by its properties, described in the sequel of this section.
A name in the metalanguage is said to be evaluated by the processor. This evaluation process is determined by the application of:

a. A fixed set of built-in rules. These rules are described below.
b. A dynamically varying list of rules, called the list of truths V, which is initially empty and which is filled during the evaluation of a name with the results of the evaluations of the simple names which constitute the name concerned.

2.3.1. The evaluation of a name.

The evaluation of a name is performed by the following process:
Step 1: The first simple name of the name is considered;

Step 2: The value of the considered simple name is determined (2.3.2);

Step 3: If the considered simple name is followed by the metasymbol co, then its value is added to V, preceded, unless V is empty, by co as a separator, and the remaining name is evaluated;

Step 4: Otherwise, the value of the considered simple name is added to V, preceded, unless V is empty, by co as a separator.

2.3.2. The value of a simple name.

The value of tr is tr.

The value of a metastring is the list of metaexpressions which is obtained by deleting the outermost "{" and "}" from the metastring.

The value of a simple term is determined by the following process:

Step 1: If the simple term contains a simple primary (2.1, rule 26), then step 2 is taken, otherwise step 3 is taken;

Step 2: The simple primary is replaced by the value of the terminal sequence of the simple primary. If the resulting sequence is a simple term, then step 1 is repeated with this simple term; otherwise, its value is undefined;

Step 3: V is applied to the determination of the value of the simple term (2.3.5).

For the definition of the application of V we need two concepts, viz. that of envelope and that of applicability.

2.3.3. The concept of envelope.

A left part can be an envelope of a simple sequence. This concept is defined in two stages.

First the case is considered that the left part is a metasequence μ and the simple sequence is a terminal sequence τ.

Let $\mu = A_1 A_2 \dots A_m$, $m \geq 1$, and $\tau = a_1 a_2 \dots a_n$, $n \geq 1$. Let m_0 be the number of (indexed) optional metavariables and let $\underline{m} = m - m_0$.

A partition of τ, $\tau = \tau_1 \tau_2 \dots \tau_m$, is defined by a selection of m-1 integers $j_1, j_2, \dots, j_{m-1}$, with $0 = j_0 \leq j_1 \leq \dots \leq j_{m-1} \leq j_m = n$, such that $\tau_i = a_{j_{i-1}+1} \dots a_{j_i}$, for $i = 1, 2, \dots, m$. If $j_{i-1}+1 > j_i$, then τ_i is defined to be the empty sequence. An ordering $<$ is defined on the partitions as follows: Let π_i, $i = 1,2$, be two partitions with associated integers $j_1^{(i)}, j_2^{(i)}, \ldots, j_{m-1}^{(i)}$. Then $\pi_1 < \pi_2$ if and only if there exists an integer p, $1 \leq p \leq m$, such that $j_p^{(1)} < j_p^{(2)}$ and $j_q^{(1)} = j_q^{(2)}$ for all $q < p$.

The following process is now applied in order to establish whether μ is an envelope of τ:

Step 1: If $n < \bar{m}$, then μ is not an envelope of τ;

Step 2: Otherwise, the first partition of τ in the given ordering is considered;

Step 3: Let $\tau = \tau_1 \tau_2 \ldots \tau_m$ be the considered partition. τ_i is said to correspond to A_i. The following relations are verified for all $i,j = 1, 2, \ldots, m$:

a. If A_i is a terminal symbol, then $\tau_i = A_i$; otherwise, if A_i is a (indexed) simple metavariable, then τ_i <u>in</u> $\bar{A}_i$ has the value <u>tr</u>; otherwise, if A_i is an (indexed) optional metavariable, then either τ_i is empty, or τ_i <u>in</u> $\tilde{\tilde{A}}_i$ has the value <u>tr</u>.

b. If τ_i and τ_j correspond to similar indexed metavariables, then they are equal.

If both relations hold then μ is an envelope of τ; otherwise, if there is a next partition of τ, then this is considered and step 3 is taken again; otherwise, μ is not an envelope of τ.

Next the general case is considered.

A left part λ is an envelope of a simple sequence σ if either

a. λ is a metasequence, σ is a terminal sequence and the above given definition holds,

or λ and σ have the following properties:

b1. λ is not a metasequence and τ is not a terminal sequence,

b2. the metasequence of λ is an envelope of the terminal sequence of σ,

b3. the simple metavariables of λ and σ are equal.

If λ is an envelope of σ, then σ is called a specific case of λ.

2.3.4. The concept of applicability.

A truth can be applicable to a simple sequence.

tr is applicable to no simple sequence.

The general form of a truth θ different from tr, is:

<CONDITION> im <LEFT PART> is <RIGHT PART>.

Truths of the form

<CONDITION> im <LEFT PART>,
<LEFT PART> is <RIGHT PART> or
<LEFT PART>

are treated respectively as:

<CONDITION> im <LEFT PART> is tr,
tr im <LEFT PART> is <RIGHT PART> or
tr im <LEFT PART> is tr.

θ is applicable to the simple sequence σ if both

a. the left part λ of θ is an envelope of σ;

b. the condition γ of θ is satisfied.

In order to establish whether γ is satisfied, the "derived condition" γ^* is constructed as follows: Let μ be the metasequence of λ, and τ the terminal sequence of σ.

Each indexed metavariable in γ which is similar to some indexed metavariable in μ is replaced by the subsequence of τ which corresponds to that indexed metavariable. Then γ is satisfied if either

a. γ^* = tr, or

b. γ^* is a terminal sequence which has the value tr, or

c. γ^* is a metasequence which is an envelope of a truth θ_0 in V (here the truths in V are searched in the order defined in 2.3.5).

Suppose θ is indeed applicable to σ. The "derived right part" ρ^* is constructed from the right part ρ of θ as follows: Each indexed metavariable in ρ which is similar to some indexed metavariable in μ or in γ^* is replaced by the corresponding subsequence of τ or θ_0 respectively.

2.3.5. The application of V.

The truths in V are ordered in the following way:
Let V be θ_1 co θ_2 co ... co θ_n. Then $\theta_i \leq \theta_j$ if and only if $i \geq j$.
The list of truths V is applied to the determination of the value of a simple sequence σ as follows:

Step 1: If V is empty, then the value of σ is σ; otherwise, step 2 is taken;

Step 2: The first truth in the given ordering is considered;

Step 3: If the considered truth θ is applicable to σ then the value of σ is the result of the simple evaluation (see below) of the derived right part of θ; otherwise, step 4 is taken;

Step 4: If there is a next truth in the given ordering, then it is considered and step 3 is repeated; otherwise, the value of σ is σ.

The result of the simple evaluation of the empty sequence is undefined.
The result of the simple evaluation of a simple name is the value of that simple name.
The result of the simple evaluation of a derived right part of the form {<NAME>} is equal to the result of applying the process defined in 2.3.1 to the name concerned, where step 4 of that process is omitted.

Remark: In the sequel, we shall not always strictly adhere to the terminology which has been introduced in this section.
By 2.3.1, the word "evaluate" refers to a process consisting of two parts:
a. The determination of the values of a list of simple names.
b. The addition of these values to V.
However, we shall use the word "evaluation" also for the determination of the value of a simple name, which is not followed by the addition of this value to V.
From the above given definitions it follows that the addition of the value of a simple name to V is omitted in the following four cases:
a. The simple name is a derived condition.
b. It is a terminal sequence of a simple primary.
c. It is generated in 2.3.3, step 3, case a.

d. It is the first (in the order given in 2.3.5) element in the list of simple names of a derived right part.

Therefore, it will be clear from the context which use of the word "evaluation" is meant.

Also, we will use "evaluation", where we mean "simple evaluation".

2.4. Semantical examples

2.4.1. Examples of the concept of envelope.

Suppose V has at a given moment the following content:

θ1: a in <letter> co

θ2: b in <letter> co

θ3: <letter> in <identifier> co

θ4: <identifier><letter> in <identifier>

In this and subsequent examples we have numbered the truths in order to make it easier to refer to them in our comments. Actually, however, these numbers do not occur in V.

Clearly, the above given list of truths is nothing but a transcription of the following grammar in Backus notation:

<letter>::= a | b

<identifier>::= <letter>|<identifier><letter>.

Given this content of V, the following relations hold:

<identifier> is an envelope of aba,

<identifier> + <identifier> is an envelope of ab + ba,

<identifier1><identifier1> is an envelope of abab but not of abba,

<identifier1><identifier><identifier1> is an envelope of abab and of abaaab (in the first case, the partition which gives this result is: $\tau = \tau_1 \tau_2 \tau_3$, where τ = abab, τ_1 = ab, τ_2 is the empty sequence and τ_3 = ab, and in the second case $\tau = \tau_1 \tau_2 \tau_3$, with τ = abaaab, τ_1 = ab, τ_2 = aa, τ_3 = ab).

<identifier><identifier><identifier> is an envelope of abaaab (here the "succesful" partition is $\tau = \tau_1 \tau_2 \tau_3$, τ_1 = a, τ_2 = b, τ_3 = aaab).

<identifier1><identifier2> is an envelope of abab and of abba.

<identifier><letter> in <identifier> is an envelope of bbb in

<identifier>, but not of bbb in <letter>, since in the second case the simple metavariables after in, i.e. <identifier> and <letter>, are not equal.

We treat the first example in more detail. <identifier> is an envelope of aba, if aba in <identifier> has the value tr. Thus, V is applied to the evaluation of aba in <identifier>. θ_4 is considered. τ = aba is first partitioned into $\tau = \tau_1\ \tau_2$, τ_1 = a, τ_2 = ba. a in <identifier> has the value tr, by applying θ_3 and θ_1. In fact, the truth θ_3: <letter> in <identifier> is treated as: tr im <letter> in <identifier> is tr, the left part of this truth is an envelope of a in <identifier> (since a in <letter> has the value tr by θ_1, and the simple metavariables after in are equal), and the condition is satisfied. Thus, the value of a in <identifier> is the value of tr, and the value of tr was defined to be tr. However, ba in <letter> does not have the value tr (θ_3 and θ_4 are not applicable, since the simple metavariables after in are different from <letter>; that θ_2 and θ_1 are not applicable follows from step 1 in the definition of 2.3.3). Thus, we conclude that the partition τ_1 = a, τ_2 = ba, is not succesful. Therefore, the next partition is considered: τ_1 = ab, τ_2 = a. ab in <identifier> has the value tr by applying θ_4, θ_3, θ_1 and θ_2. a in <letter> has the value tr by θ_1. Consequently, θ_4 is applicable to aba in <identifier>, and we find that aba in <identifier> has the value tr, which means that <identifier> is an envelope of aba.

2.4.2. Examples of the evaluation of a name (see also section 4.2).

2.4.2.1. The Euclidean algorithm for the greatest common divisor (4.2, example 1).

Let V consist of the following list of truths:

θ_1: 1 <integer> in <integer> co

θ_2: (<integer1>, <integer1><integer2>) is (<integer1>, <integer2>) co

θ_3: (<integer1><integer2>, <integer2>) is (<integer1>, <integer2>) co

θ_4: (<integer1>, <integer1>) is <integer1>

Then for each pair of natural numbers (n,m):
The result of applying V to the evaluation of $(\overline{n},\overline{m})$ is $\overline{gcd(n,m)}$, where $\overline{n}$ stands for a sequence of n symbols 1.

Examples (in an obvious notation):

$$(\overline{42},\overline{105}) \overset{\theta_2}{\Longrightarrow} (\overline{42},\overline{63}) \overset{\theta_2}{\Longrightarrow} (\overline{42},\overline{21}) \overset{\theta_3}{\Longrightarrow} (\overline{21},\overline{21}) \overset{\theta_4}{\Longrightarrow} \overline{21}$$

$$(\overline{9},\overline{2}) \overset{\theta_3}{\Longrightarrow} (\overline{7},\overline{2}) \overset{\theta_3}{\Longrightarrow} (\overline{5},\overline{2}) \overset{\theta_3}{\Longrightarrow} (\overline{3},\overline{2}) \overset{\theta_3}{\Longrightarrow} (\overline{1},\overline{2}) \overset{\theta_2}{\Longrightarrow} (\overline{1},\overline{1}) \overset{\theta_4}{\Longrightarrow} \overline{1}$$

V defines the Euclidean algorithm with repeated subtraction instead of division. The subtraction is automatically performed by the partitioning mechanism of the envelope concept as a result of the requirement that subsequences corresponding to similar indexed metavariables be equal.

2.4.2.2. Definition of lexicographical ordering (4.2, example 2).

Let V consist of the following list of truths:

θ1 : a in <letter> co

θ2 : b in <letter> co

θ3 : c in <letter> co

θ4 : d in <letter> co

θ5 : e in <letter> co

θ6 : <letter><word> in <word> co

θ7 : <word> pre <word> is false co

θ8 : <letter1> pre <letter2> im <letter1><word> pre <letter2><word> co

θ9 : <letter1><word1> pre <letter1><word2> is <word1> pre <word2> co

θ10: <letter1><word> pre <letter1> is false co

θ11: <letter1> pre <letter1><word> co

θ12: <letter2> pre <letter3> im <letter1> pre <letter3>
is <letter1> pre <letter2> co

θ13: <letter> pre a is false co

θ14: a pre b co

θ15: b pre c co

θ16: c pre d co

θ17: d pre e co

θ18: <letter1> pre <letter1>

For each two words w_1, w_2 over the alphabet $\{a,b,c,d,e\}$, w_1 pre w_2 has the value tr if w_1 lexicographically precedes w_2, otherwise w_1 pre w_2 has the value false.

Example: the evaluation of dbc pre dee.

By the first applicable truth, θ_9, the value of dbc pre dee is the value of bc pre ee. The left part of θ_8 is an envelope of bc pre ee. Therefore, θ_8 is applicable to bc pre ee, provided the derived condition, viz. b pre e, has the value tr. The left part of θ_{12} is an envelope of b pre e. Thus, θ_{12} is applicable to b pre e, if the derived condition, <letter2> pre e, is satisfied. This derived condition is not a terminal sequence. Consequently, the list of truths is searched for a truth which is enveloped by <letter2> pre e. θ_{17} is such a truth.Hence, application of θ_{12} to b pre e leads to the evaluation of the derived right part b pre d, where b is the subsequence corresponding to <letter1> and d the subsequence corresponding to <letter2>. By the same process it is found that the value of b pre d is the value of b pre c, which has the value tr by θ_{15}. Thus, θ_8 is found to be applicable to bc pre ee, and the value of bc pre ee is tr. The final result is therefore that dbc pre dee has the value tr.

CHAPTER 3

PROPERTIES OF THE METALANGUAGE

In this chapter we give some basic results on the relation between the metalanguage and two subjects in the theory of formal languages.
In section 1 we consider several definitions of computability, viz. Markov algorithms, Turing machines and recursive functions, and we prove that every function which is computable by means of one of these systems is computable in terms of the metalanguage (a more precise formulation is given below). Since it is well known that the three systems are equivalent, it would have been sufficient to give this proof for anyone of the definitions. However, we treat each case separately in order to have more examples to illustrate the various concepts of the metalanguage.
In section 2 we make some remarks on the connection between the metalanguage and a few aspects of the theory of phrase structure languages.

In the sequel, it will be convenient to use the following terminology:
If a name in the metalanguage has the form
$\{$<LIST OF METAEXPRESSIONS>$\}$ <u>co</u> <SIMPLE TERM>,
then we consider the list of metaexpressions as a "metaprogram" for the simple term. This is explained by the fact that the list of metaexpressions is left unchanged when it is added to V, whereas in the evaluation of the simple term we use the list of metaexpressions. When we apply the list of metaexpressions, say V_0, to the simple term σ, we say that σ is evaluated by means of V_0 and we denote the result by $V_0(\sigma)$.
Moreover, we introduce the following notation: An "alphabet" A is any finite non empty set; the elements of A are called "symbols". A^* denotes the set of all finite sequences of elements of A, including the empty sequence. The elements of A^* are called "words" over A, the empty word

is denoted by ε. For any $a \in A$, and for any integer $n \geq 0$, a^n denotes the sequence of n symbols a.

3.1. Definitions of computability

3.1.1. Markov algorithms (for notations see [37]; cf. also 4.2.2, example 5).

Theorem: Let $A = \{a_1, a_2, \ldots, a_n\}$ be an alphabet and let $\mathfrak{A}$: $P_1 \rightarrow (\cdot)Q_1$, $P_2 \rightarrow (\cdot)Q_2$, ..., $P_n \rightarrow (\cdot)Q_n$ be the scheme of a normal algorithm in A. ($\rightarrow (\cdot)$ stands for either $\rightarrow$ or $\rightarrow\cdot$). Let α be an arbitrary symbol not in A. Then there exists a metaprogram V_0 such that for each word $w_0 \in A^*$ to which $\mathfrak{A}$ is applicable: $\alpha\,\mathfrak{A}(w_0) = V_0(\alpha w_0)$.

Proof: We define three lists of metaexpressions, V_1, V_2 and V_3. For the set of terminal symbols we choose $A \cup \{\alpha\}$ [1].

1. V_1 is the list
   a_1 in <symbol> co a_2 in <symbol> co ... co a_n in <symbol>
2. V_2 is the list
   <symbol><tape> in <tape>
3. For each substitution formula $P_i \rightarrow Q_i$ of $\mathfrak{A}$ we define an associated truth T_i as:
   α <tape1> P_i <tape2> is α <tape1> Q_i <tape2>.
   For each substitution formula $P_j \rightarrow\cdot\, Q_j$ we define an associated T_j as:
   α <tape1> P_j <tape2> is {α<tape1> Q_j <tape2>}
   V_3 is defined as T_n co T_{n-1} co ... co T_1.

Then we define V_0 as V_1 co V_2 co V_3. The proof of the assertion now follows from the following points:

1. According to Markov's definition, a left hand member P_i of a substitution formula $P_i \rightarrow (\cdot)Q_i$ enters into a word $w \in A^*$ if and only if w has the form $w = uP_iv$, with $u, v \in A^*$. This is equivalent to our

---

1) This set is not a subset of the set of terminal symbols, given in chapter 4. However, it is easy to define a mapping from $A \cup \{\alpha\}$ into this set, for example, a_1 corresponds to al, α to alpha, etc.

definition of envelope, where we require that there exist a partition of w, $w = w_1\ w_2\ w_3$, such that <tape1> is an envelope of w_1, $P_i = w_2$, and <tape2> is an envelope of w_3.

2. Markov's requirement of selecting the first entry corresponds to our requirement of selecting the smallest partition.
3. Markov's definition of the way in which V is applied to the transformation of a word w, i.e. by first trying to apply $P_1 \rightarrow (\cdot)Q_1$, in case of success continuing with the transformed word, where P_1 is replaced by Q_1, otherwise by trying to apply $P_2 \rightarrow (\cdot)Q_2$, etc., is the same as our way of applying V
4. In Markov's definition the process is stopped if one meets the symbol $\rightarrow\cdot$, while in our definition the value of a metastring is also found immediately and not by applying V again.
5. In Markov's definition, if none of the substitution formulae is applicable to w, the result of applying $\mathfrak{A}$ to w is w itself. The same holds for the evaluation of w by means of V_0.
6. From 3,4 and 5 it follows that the evaluation of w by means of $\mathfrak{A}$ terminates if and only if the evaluation of w by means of V_0 terminates.
7. We have introduced the extra symbol α in order to ensure that the length of the sequence which is evaluated is always ≥ 1. This is necessary because in Markov's definition it is possible that w_0 or one of its transforms is empty, whereas the evaluation of the empty sequence in the metalanguage is undefined.

Apparently the metalanguage can be considered as an extension of Markov algorithms in the sense that every basic concept of Markov's system is contained in the metalanguage. The main extra features of our system are:

1. The use of metavariables.
2. The possibility of dynamically adding new truths.
3. The use of a condition in the truths.

3.1.2. Turing machines

In this section we use the terminology of Davis [16] except for his use of the term "internal configuration", which we replace by "state". Cf. also 4.2.2, example 6.

Theorem: Let Z be a simple Turing machine. There exists a metaprogram V_0 such that for each instanteneous description α, $V_0(\alpha) = Res_Z(\alpha)$, where $Res_Z(\alpha)$ is the resultant of α with respect to Z.

Proof: Let $\Sigma = \{S_0, S_1, \ldots, S_n\}$ be the alphabet of Z, and Q = $= \{q_1, q_2, \ldots, q_m\}$ the set of states of Z. Let Z be the set of quadruples $\{q_{i_1} S_{j_1} P_{k_1} q_{l_1}, \ldots, q_{i_r} S_{j_r} P_{k_r} q_{l_r}\}$, with $S_{j_p} \in \Sigma$, $q_{i_p}, q_{l_p} \in Q$ and $P_{k_p} \in \Sigma \cup \{R,L\}$, for p = 1, 2, ..., r.

We define five lists of metaexpressions $V_1, V_2, \ldots, V_5$. (In this and the following proofs we do not explicitly list the set of terminal symbols, since this can be obtained easily from the construction of V_0.)

1. V_1 is the list

   S_0 in <symbol> co S_1 in <symbol> co ... co S_n in <symbol>

2. V_2 is the list

   <symbol><tape> in <tape>

3. V_3 is the list

   q_1 in <state> co q_2 in <state> co ... co q_m in <state>

4. V_4 is the list $T_{4,1}$ co $T_{4,2}$ co $T_{4,3}$ co $T_{4,4}$ co $T_{4,5}$, where

   $T_{4,1}$ is <state1><symbol1><symbol2><state2> im
   <tape1><state1><symbol1><tape2> is
   <tape1><state2><symbol2><tape2>

   $T_{4,2}$ is <state1><symbol1> R <state2> im
   <tape1><state1><symbol1><tape2> is
   <tape1><symbol1><state2><tape2>

   $T_{4,3}$ is <state1><symbol1> R <state2> im
   <tape1><state1><symbol1> is
   <tape1><symbol1><state2> S_0

   $T_{4,4}$ is <state1><symbol1> L <state2> im
   <tape1><symbol2><state1><symbol1><tape2> is
   <tape1><state2><symbol2><symbol1><tape2>

   $T_{4,5}$ is <state1><symbol1> L <state2> im
   <state1><symbol1><tape1> is
   <state2> S_0 <symbol1><tape1>

5. V_5 is defined to be the list of quadruples of Z, separated by metacommas.

V_0 is defined as V_1 co V_2 co ... co V_5.
Let α, β be two instanteneous descriptions. The proof of the assertion now follows from Davis' definition of the relation α → β ([16], Ch.1, def. 1.7).

There are five possibilities:

1. There exist tape expressions $P,Q \in \Sigma^*$, such that $\alpha = Pq_iS_jQ$, $\beta = Pq_1S_kQ$, and Z contains $q_iS_jS_kq_1$. This means that $T_{4,1}$ is applicable to α, since
   a. <tape1> is an envelope of P by applying the truths in V_1 and V_2,
   b. <state1> is an envelope of q_i, by V_3,
   c. <symbol1> is an envelope of S_j, by V_1,
   d. <tape2> is an envelope of Q, by V_1 and V_2,
   e. The derived condition is q_iS_j <symbol2><state2>,
   f. q_iS_j <symbol2><state2> is an envelope of the truth $q_iS_jS_kq_1$, which is one of the truths in V_5.

   Thus, the condition of $T_{4,1}$ is satisfied, and the value of Pq_iS_jQ is the value of the derived right part of $T_{4,1}$; i.e., the value of Pq_1S_kQ, where this derived right part is constructed as follows:
   a. <tape1>, which also occurs in the left part of $T_{4,1}$, is replaced by P,
   b. <state2>, which also occurs in the derived condition, is replaced by q_1,
   c. <symbol2> which also occurs in the derived condition, is replaced by S_k
   d. <tape2>, which occurs in the left part, is replaced by Q.
2. $\alpha = Pq_iS_jS_kQ$, $\beta = PS_jq_1S_kQ$, and Z contains $q_iS_jRq_1$. By applying $T_{4,2}$ it follows in the same way that the value of α is the value of β.

3, 4, and 5 are treated similarly.

Finally, if α is terminal, then none of the truths in V_0 is applicable to it

3.1.3 Recursive functions.

In this section we use the terminology of Mendelson [37].

Theorem: There exists a metaprogram V_0 such that for each partial recursive function f of n arguments and for each n-tuple $(x_1, x_2, \ldots, x_n)$ for which f is defined the following holds: Let ϕ be the notation in the metalanguage for the function f, and ξ for the integer list $x_1, x_2, \ldots, x_n$. (This notation is introduced in the proof.)
Then: $V_0(\phi(\xi)) = f(x_1, x_2, \ldots, x_n)$.

Proof: We define nine lists of metaexpressions:

1. Syntactic definition of integer and integer list (V_1):

   1 <integer> in <integer> co
   <integer> in <integer list> co
   <integer>, <integer list> in <integer list>

   A sequence of n symbols 1 denotes the integer n-1.

2. Syntactic definition of the initial functions (V_2):

   Z in <function> co
   N in <function> co
   U <integer> 1 in <function>

3. Syntactic definition of the rules for obtaining new functions from given functions by means of substitution, recursion, and the μ-operator (V_3):

   <function>(<function list>) in <function> co
   ρ <function><function> in <function> co
   μ <function> in <function>

4. Syntactic definition of function list (V_4):

   <function> in <function list> co
   <function>, <function list> in <function list>

5. Definition of the value of the initial functions (V_5):

   Z(<integer list>) is 1 co
   N(<integer1>) is <integer1> 1 co
   U <integer1> 1 (<integer1>, <integer list1>) is
   U <integer1> (<integer list1>) co
   U 11 (<integer1>) is <integer1> co
   U 11 (<integer1>, <integer list>) is <integer1>

6. Definition of the result of substituting a list of functions in a function (V_6):

<function1>(<function list1>)(<integer list1>) is
<function1>(va{<function list1>(<integer list1>)}) co
<function1>, <function list1>(<integer list1>) is
va{<function1>(<integer list1>)}, va{<function list1>(<integer list1>)}

7. Definition of recursion (V_7):

ρ <function1><function2>(<integer list1>, <integer1> 1) is
<function2>(<integer list1>, <integer1>,
va{ρ<function1><function2>(<integer list1>,<integer1>)}) co
ρ <function1><function>(<integer list1>, 1) is <function1>(<integer list1>)

8. Definition of equality to zero (V_8):

1 = 1 co
<function1>(<integer list1>) = 1 is va{<function1>(<integer list1>)} = 1

9. Definition of the μ-operator (V_9):

$V_9 = T_{9,1}$ co $T_{9,2}$ co $T_{9,3}$, where
$T_{9,1}$ is
μ <function1>(<integer list1>) is μ <function1>(<integer list1>) : 1,
$T_{9,2}$ is
μ <function1>(<integer list1>) : <integer1> is
μ <function1>(<integer list1>) : <integer1> 1,
$T_{9,3}$ is
<function1>(<integer list1>, <integer1>) = 1 im
μ <function1>(<integer list1>) : <integer1> is <integer1>

$T_{9,3}$ tests whether <integer list1>, <integer1> is a zero of <function1>. If this is not the case, then <integer1> is increased by one by applying $T_{9,2}$, and $T_{9,3}$ is tried again. This process must terminate since f was defined for $(x_1, x_2, \ldots, x_n)$.

V_0 is defined as the list V_1 co V_2 co ... co V_9.
The proof now follows from the construction of V_0.

3.2. Phrase structure languages and the metalanguage

In this section we first recall the definition of a phrase structure language, we define Chomsky's type 3, type 2, type 1, and type 0 languages and we introduce the various abstract machines which define the different types of languages. Then we prove that for each type 0 language there exists a metaprogram which generates this language. Next we investigate the ressemblance between our notion of envelope and the way in which one recognizes whether a word belongs to a context free language. Then we give a simple example of the recognition of a context sensitive language and finally we exhibit definitions in terms of the metalanguage of the above mentioned abstract machines.

3.2.1. Definition of phrase structure languages.

The definitions in this section follow Ginsburg [22].

A phrase structure grammar is a 4-tuple $G = (V, \Sigma, P, \sigma)$, where

1. V is an alphabet,
2. $\Sigma \subset V$ is an alphabet (the set of terminal symbols),
3. P is a finite set of ordered pairs (u,v), $u \in (V \setminus \Sigma)^* - \{\varepsilon\}$, $v \in V^*$,
4. $\sigma \in V - \Sigma$.

The elements of $V - \Sigma$ are called (metalinguistic) variables. The elements (u,v) of P are usually written $u \to v$.

Let $G = (V, \Sigma, P, \sigma)$ be a phrase structure grammar.
For $w,y \in V^*$, we write $w \implies y$, if there exist z_1, z_2, u, $v \in V^*$, such that $w = z_1 u z_2$, $y = z_1 v z_2$, and $u \to v \in P$.
For $w,y \in V^*$, we write $w \overset{*}{\implies} y$, if either $w = y$, or there exist $w_0 = w$, w_1, w_2, ..., $w_r = y$ such that $w_i \implies w_{i+1}$ for $i = 0, 1, \ldots, r-1$.
If $G = (V, \Sigma, P, \sigma)$ is a phrase structure grammar then the subset $L(G) = \{w \in \Sigma^* \mid \sigma \overset{*}{\implies} w\}$ of Σ^* is called a phrase structure language.
A phrase structure language is called ε-free if it does not contain the empty word.
Remark: In the remainder of this chapter we restrict ourselves to ε-free languages. This is only a matter of convenience, since, by using a device as in theorem 3.1.1, it would have been easy to avoid it.

Each phrase structure grammar is called "of type 0".

A phrase structure grammar $G = (V, \Sigma, P, \sigma)$ is called of type 1 or context sensitive if all elements of P have the form $u\xi v \to uyv$, $u,v \in (V - \Sigma)^*$, $\xi \in V - \Sigma$ and $y \in V^* - \{\varepsilon\}$.

G is called of type 2 or context free if all elements of P have the form $\xi \to v$, $\xi \in V - \Sigma$, $v \in V^*$

G is called of type 3 if it is either left linear or right linear; it is left linear (right linear) if all elements of P have the form $\xi \to u$ or $\xi \to \nu u$ ($\xi \to u\nu$), with $\xi, \nu \in V - \Sigma$, $u \in \Sigma^*$.

A phrase structure language L is called of type i, i = 0, 1, 2, 3, if i is the largest number such that there exists a grammar G of type i such that $L = L(G)$.

A finite automaton is a 5-tuple $A = (K, \Sigma, \delta, q_0, F)$, where

1. K is a finite non empty set (of "states"),
2. Σ is an alphabet (of "inputs"),
3. δ is a mapping from $K \times \Sigma$ into K (the "next state function")
4. $q_0 \in K$ (the "initial state"),
5. $F \subset K$ (the set of "final states").

δ is extended to $K \times (\Sigma^* - \{\varepsilon\})$ as follows:

$\delta(q, aw) = \delta(\delta(q,a),w)$, where $q \in K$, $a \in \Sigma$ and $w \in \Sigma^* - \{\varepsilon\}$.

Let A be a finite automaton. Then

$T(A) = \{w \in \Sigma^* - \{\varepsilon\} \mid \delta(q_0,w) \in F\}$.

T(A) is the set of words "accepted" by A.

A pushdown automaton is a 7-tuple $M = (K, \Sigma, \Gamma, \delta, z_0, q_0, F)$, where

1. K, Σ, q_0, F are defined as for finite automata,
2. Γ is a finite non empty set (of "pushdown symbols"),
3. $z_0 \in \Gamma$ (the "initial pushdown symbol"),
4. δ is a mapping from $K \times \Sigma \times \Gamma$ into the set of all finite subsets of $K \times \Gamma^*$.

We define the relations $\vdash$ and $\vdash^*$ as follows:

For $q_1, q_2 \in K$, $a \in \Sigma$, $w \in \Sigma^*$, $\alpha \in \Gamma^*$, $z \in \Gamma$, $\gamma \in \Gamma^*$,

$(q_1, aw, z\alpha) \vdash (q_2, w, \gamma\alpha)$, if $\delta(q_1, a, z)$ contains (q_2, γ).

For $p, q \in K$, $w \in \Sigma^*$, $a_i \in \Sigma$ $(1 \leq i \leq k)$, $\alpha, \beta \in \Gamma^*$,
$(q, w, \alpha) \models^* (q, w, \alpha)$ and
$(p, a_1 a_2 \ldots a_k w, \alpha) \models^* (q, w, \beta)$ if there exist $p_1 = p, p_2, \ldots, p_{k+1} = q \in K$, and $\alpha_1 = \alpha, \alpha_2, \ldots, \alpha_{k+1} = \beta \in \Gamma^*$ such that
$(p_i, a_i a_{i+1} \ldots a_k w, \alpha_i) \vdash (p_{i+1}, a_{i+1} a_{i+2} \ldots a_k w, \alpha_{i+1})$, for $1 \leq i \leq k$.

Let M be a pushdown automaton. Then
$T(M) = \{w \in \Sigma^* - \{\varepsilon\} \mid (q_0, w, z_0) \models^* (q, \varepsilon, \alpha)$ for some $q \in F$ and $\alpha \in \Gamma^*\}$.
T(M) is the set of words "accepted" by M.

The following theorems are known: Let Σ be an alphabet.

1. A subset L of Σ^* is a type 3 language if and only if it is accepted by some finite automaton [13].
2. A subset L of Σ^* is a type 2 language if and only if it is accepted by some pushdown automaton [12].
3. A subset L of Σ^* is a type 1 language if and only if it is accepted by some linear bounded automaton
   (for the definition of linear bounded automata and the proof of this theorem see Kuroda [29]).
4. A subset L of Σ^* is a type 0 language if and only if it is generated by some Turing machine (see 3.2.2).

3.2.2. Type 0 and type 2 languages.

Theorem: Each type 0 language can be defined by means of the meta-language.

Proof: Follows immediately from theorem 3.1.3 and the fact that each type 0 language is a recursively enumerable set [11].

Since context free languages form a subclass of the class of all phrase structure languages, this theorem also holds for context free languages. However, we give a separate proof of this special case, because

a. This case can be proved directly, i.e., without using recursive functions.
b. The proof illustrates the relation between the concept of envelope and the way in which one recognizes whether a word belongs to a context free language.

Theorem: Let L be an ε-free context free language. Let $G = (V, \Sigma, P, \sigma)$ be a grammar for L. Then there exists a metaprogram V_0 such that for each $w \in \Sigma^* - \{\varepsilon\}$:

$w \in L$ if and only if V_0 (w <u>in</u> <σ>) = <u>tr</u>.

Proof: We construct a grammar $G' = (V', \Sigma, P', \sigma)$ such that

1. $L(G') = L(G)$.
2. The rules of P' have either the form $A \rightarrow BC$ or $D \rightarrow d$ $(A, B, C, D \in V' - \Sigma,\ d \in \Sigma)$.

For this construction see e.g. [11].

With each rule in P' we associate a truth as follows:

If the rule has the form $A \rightarrow BC$, then the associated truth is <B><C> <u>in</u> <A>.

If the rule has the form $D \rightarrow d$, the associated truth is d <u>in</u> <D>.

V_0 is defined as the list of truths which are associated with the rules in P'.

We now prove: For each $A \in V' - \Sigma$ and each $w \in \Sigma^* - \{\varepsilon\}$, it follows that $A \overset{*}{==>} w$ if and only if V_0 (w <u>in</u> <A>) = <u>tr</u>. By considering the special case $A = \sigma$, the theorem follows immediately from this equivalence.

1. Let $A \in V' - \Sigma$ and $w \in \Sigma^* - \{\varepsilon\}$. Suppose $A \overset{*}{==>} w$. We prove that V_0 (w <u>in</u> <A>) = <u>tr</u>, by induction on the length of w.
   a. Suppose w has length 1, i.e. $w = a \in \Sigma$. $A \overset{*}{==>} w$ is necessarily a derivation of length 1, i.e. $A \overset{*}{==>} w$ is simply $A ==> w(=a)$. This means that $A \rightarrow a \in P'$, whence a <u>in</u> <A> $\in V_0$. Therefore, a <u>in</u> <A> has the value <u>tr</u>.
   b. Suppose the assertion has been proved for any $B \in V' - \Sigma$ with a word w of length < n. Suppose $A \overset{*}{==>} w$ where w has length n. Then there exists $C, D \in V' - \Sigma$ such that $A ==> CD \overset{*}{==>} w$. (This follows from the special form of the grammar G'.) There exist $u, v \in \Sigma^* - \{\varepsilon\}$, such that $C \overset{*}{==>} u$, $D \overset{*}{==>} v$, and $w = uv$ ([22], lemma 1.4.6). By the induction hypothesis V_0 (u <u>in</u> <C>) = <u>tr</u>, and V_0 (v <u>in</u> <D>) = <u>tr</u>. Moreover, <C><D> <u>in</u> <A> is a truth in V_0; hence, it follows that w <u>in</u> <A> has the value <u>tr</u> by the definition of envelope.

2. Suppose V_0 (w in <A>) = tr. We prove A $\overset{*}{\Longrightarrow}$ w, again by induction on the length of w.
   a. If w has length 1, i.e. w = a $\in \Sigma$, then a in <A> $\in V_0$; hence, A $\rightarrow$ a $\in$ P', Therefore, A $\overset{*}{\Longrightarrow}$ w.
   b. Suppose the assertion has been proved for each B $\in$ V' $-\ \Sigma$ with w of length < n. Suppose V_0(w in <A>) = tr. According to the definition of envelope there is a truth in V_0 of the form <C><D> in <A>, and a partition of w, w = uv, such that V_0(u in <C>) = tr and V_0(v in <D>) = tr. By the induction hypothesis, C $\overset{*}{\Longrightarrow}$ u and D $\overset{*}{\Longrightarrow}$ v. A $\rightarrow$ CD is a rule in P' by the definition of V_0. Thus, from A $\rightarrow$ CD, C $\overset{*}{\Longrightarrow}$ u, D $\overset{*}{\Longrightarrow}$ v and w = uv it follows that A $\Longrightarrow$ CD $\overset{*}{\Longrightarrow}$ uv = w.

3.2.3. A type 1 language (cf. 4.2.2, example 7).

The set $\{a^n\ b^n\ a^n \mid n \geq 1\}$ is not a type 2 language ([22]). Therefore, we cannot use the second theorem of 3.2.2 to recognize whether a word belongs to this set. However, by using more of the mechanism of the metalanguage, it is possible to construct a metaprogram V_0 which does perform this recognition.

Let V_0 be defined as:

a <as> in <as> co b <bs> in <bs> co

aba in <ABA> co

<as1> a <bs1> b <as1> a in <ABA> is

<as1> <bs1> <as1> in <ABA>.

It is easy to see that:

1. V_0($a^p\ b^p\ a^p$ in <ABA>) = tr, for each $p \geq 1$.
2. For $p \neq q$, V_0($a^p\ b^q\ a^p$ in <ABA>) = $a^{p_1}\ b^{q_1}\ a^{p_1}$ in <ABA>, where $p_1 = p + 1 - \min(p,q)$, $q_1 = q + 1 - \min(p,q)$.
3. V_0(w in <ABA>) = w in <ABA> for each other word w $\in \{a,b\}^* - \varepsilon$.

3.2.4. Definition of abstract machines.

In this section we show how to define each of the four abstract machines that define the type 3, 2, 1, and 0 phrase structure languages in terms of the metalanguage.

3.2.4.1. Finite automata (cf. 4.2.2, example 8).

Theorem: Let $A = (K, \Sigma, \delta, q_0, F)$ be a finite automaton.
There exists a metaprogram V_0 such that for each $w \in \Sigma^* - \{\varepsilon\}$:
$V_0(q_0w) =$ *tr* if and only if $w \in T(A)$.

Proof: Let $K = \{q_0, q_1, \ldots, q_n\}$, $\Sigma = \{a_1, a_2, \ldots, a_m\}$, and
$F = \{q_{i_1}, q_{i_2}, \ldots, q_{i_r}\}$.
We define six lists of metaexpressions:

1. V_1, V_2, and V_3 are defined as in the proof of theorem 3.1.2
2. V_4 is defined as
   q_{i_1} *in* <final state> *co* q_{i_2} *in* <final state> *co* ... *co* q_{i_r} *in* <final state>
3. V_5 is the list
   <state1><symbol1><state2> *im*
   <state1><symbol1><tape1> *is* <state2><tape1> *co*
   <final state>
4. For each $\delta(q_i, a_j) = q_k$, we define an associated truth $q_i\ a_j\ q_k$. V_6 is the list of these truths.

Let V_0 be V_1 *co* V_2 *co* ... *co* V_6. The proof now follows from an argument similar to that used in the proof of theorem 3.1.2.

Remark: The notation used in 4.2.2, example 8, differs slightly from the one used in this proof.

3.2.4.2. Pushdown automata (cf. 4.2.2, example 9).

Theorem: Let $M = (K, \Sigma, \Gamma, \delta, z_0, q_0, F)$ be a pushdown automaton. There exists a metaprogram V_0 such that for each $w \in \Sigma^* - \{\varepsilon\}$ we have: $w \in T(M)$ if and only if $V_0(q_0\ w\ z_0)$ contains the metasymbol *tr*.

Proof: Let $K = \{q_0, q_1, \ldots, q_n\}$, $\Sigma = \{a_1, a_2, \ldots, a_m\}$, $\Gamma = \{z_0, z_1, \ldots, z_p\}$ and $F = \{q_{i_1}, q_{i_2}, \ldots, q_{i_r}\}$.
V_0 is constructed from ten lists of metaexpressions (we assume that K, Σ and Γ are disjoint sets):

1. V_1, V_2, V_3 and V_4 are defined as in the proof of theorem 3.2.4.1.
2. V_5 is the list
   z_0 *in* <pd symbol> *co* ... *co* z_p *in* <pd symbol>
   "pd" is an abbrevation of "pushdown".

3. V_6 is the list
   <pd symbol><pd tape> in <pd tape>
4. V_7 is the list
   <pd tape> in <pd tapelist> co
   <pd tape>, <pd tapelist> in <pd tapelist>
5. V_8 is the list
   <state><statelist> in <statelist>
6. $V_9 = T_{9,1}$ co $T_{9,2}$ co $T_{9,3}$ co $T_{9,4}$, where
   $T_{9,1}$ is
   <state1><symbol1><pd symbol1><statelist1><pd tapelist1> im
   <state1><symbol1><tape1><pd symbol1><pd tape1> is
   <statelist1><tape1><pd tapelist1>, <pd tape1>

   $T_{9,2}$ is
   <state1><statelist1><tape1><pd tape1>, <pd tapelist1>, <pd tape2> is
   {<state1><tape1><pd tape1><pd tape2> co
   <statelist1><tape1><pd tapelist1>, <pd tape2>}

   $T_{9,3}$ is
   <state1><tape1><pd tape1>, <pd tape2> is
   <state1><tape1><pd tape1><pd tape2>

   $T_{9,4}$ is
   <statelist><final state><statelist><pd tapelist>
7. V_{10} is constructed as follows:
   With each $\delta(q_i, a_j, z_k) = \{(q_{i_1}, u_{i_1}), \ldots, (q_{i_l}, u_{i_l})\}$, where $q_i, q_{i_1}, \ldots, q_{i_l} \in K$, $a_j \in \Sigma$, $z_k \in \Gamma$, $u_{i_1}, \ldots, u_{i_l} \in \Gamma^*$, $l \geq 1$, we associate a truth $q_i\ a_j\ z_k\ q_{i_1}\ q_{i_2} \ldots q_{i_r}\ u_{i_1}, u_{i_2}, \ldots, u_{i_r}$.
   V_0 is the list of these associated truths, separated by metacommas.

Then V_0 is defined as V_1 co V_2 co ... co V_{10}.
The proof is again similar to the proof of theorem 3.1.2. The non-deterministic character of the pushdown automaton is represented by $T_{9,2}$: as a result of this truth the different courses of action which the pushdown automaton can take, corresponding to the different choices from the sets $\delta(q, a, z)$, are all treated successively. If one of these combinations leads to the value tr (by application of $T_{9,4}$) then it follows that tr occurs in $V_0(q_0\ w\ z_0)$.

Remark: Again there are some inessential differences with 4.2.2, example 9.

3.2.4.3. Linear bounded automata.

Kuroda [29] has proved that a phrase structure language is a type 1 language if and only if it is accepted by a linear bounded automaton. Essentially this is a non deterministic "Turing machine", with a finite memory; i.e., an equivalent metaprogram can be constructed for a linear bounded automaton by modifying the metaprogram which was constructed in the proof of theorem 3.1.2 as follows:

a. $T_{4,3}$ and $T_{4,5}$ are deleted, since these truths give the possibility of extending the tape indefinitely to the left and right.

b. Some truths are added which represent the fact that one now has a choice from different states for the next state. This can be done in a manner similar to the one used in $T_{9.2}$ in the metaprogram of 3.2.4.2.

3.2.4.4. Turing machines.

A set is a type 0 language if and only if it can be generated by a Turing machine. The construction of a metaprogram, equivalent to a given Turing machine, was given in 3.1.2. (Cf. also the first theorem of 3.2.2.).

CHAPTER 4

DEFINITION OF THE METALANGUAGE

In this chapter the processor is defined by an ALGOL 60 program. After this, several examples are exhibited of the evaluation of a name by the processor.

4.1. The ALGOL 60 program for the processor

First we give a general survey of the program.
We distinguish six groups of procedures:

1. The input procedures
   Init0, Init, RFS, symbol, read metavariable and read underlined symbol.
   The input/output medium used is paper tape, punched in MC flexowriter code. Heptads from the input tape are read by means of the code procedure REHEP (see group 6).
   The input procedures are defined in such a way that:
   a. A terminal symbol of the metalanguage is either a flexowriter symbol (these are listed below), or an underlined sequence of flexowriter symbols, different from each of the metasymbols.
   b. A metavariable is defined as in chapter 2, section 1.
      Thus, a metavariable is denoted by the symbol "<" or "≤", a sequence of metaletters, possibly a sequence of metadigits, and the symbol ">" or "≥" respectively.
      Due to the restricted character set on the flexowriter, we have no way of distinguishing between metaletters (metadigits) and letters (digits) which occur in the language we want to define (e.g. ALGOL 60). In ALGOL 60 this causes no special problems, since a combination like "<, sequence of letters, >" will not

occur in a syntactically correct program. If one should want to define a language in which this combination may indeed occur, one should use another denotation for the metavariables.

c. Terminal symbols, metasymbols and metavariables are represented uniquely by integers.

d. Each name is required to end with a stopcode punching (a stopcode is a punching symbol that leaves no visible mark on the typewriter sheet).

2. The output procedure output0

Heptads are punched on the output tape by means of the code procedure PUHEP (see 6).

3. The procedures

SIMPLE NAME, LIST OF METAEXPRESSIONS, SIMPLE TERM, SIMPLE FACTOR, METAEXPRESSION, LEFT PART, RIGHT PART, LIST OF SIMPLE RIGHT PARTS, SIMPLE RIGHT PART, IND METATERM, IND METAFACTOR, TERMINAL SEQUENCE, METASEQUENCE, IND METASEQUENCE, and SIMPLE METAVARIABLE.

These procedures check the syntax of the name which is offered to the processor. They reflect the rules for the syntax of the metalanguage of chapter 2, section 1.

(The technique used here was inspired by [28]).

4. The auxiliary procedures

Terminal symbol, Simple metav, Metav, Ind metav, Opt metav, Ind simple metav, Ind opt metav, Non ind metav, similar, metaletter, metadigit, error, and add to Sequence.

5. The procedures

NAME, add to V, envelope, evaluate, derive condition, derive simple right part, and derive right part.

These procedures contain the definition proper of the processor.

A call of the procedure NAME results in the determination of the value of the first simple name of the name which is offered to the processor by means of a call of the procedure evaluate, the addition of this value to V by means of a call of the procedure add to V, and, if necessary, a recursive call of NAME to treat the rest of the name.

6. The following library routines which are available without declaration in the MC ALGOL system:

read:	a function designator, assigning to its identifier the next number on the input tape.
REHEP:	an integer procedure, assigning to its identifier the value of the next heptad on the input tape.
PUHEP(f):	a procedure, punching the value of f ($0 \leq f \leq 127$) as a heptad on the output tape.
PUNLCR:	a procedure, punching a new line carriage return symbol on the output tape.
ABSFIXP(n,0,x):	a procedure, punching the absolute value of x, rounded to an integer, using n digits and replacing leading zeroes by spaces.
PUTEXT(string):	a procedure that punches the actual string on the output tape.
RUNOUT:	a procedure that punches a piece of blank tape.

Remarks:

1. The left and right metaparentheses which are defined here to be denoted by (and), are denoted in the explanatory chapters (i.e. chapters 2, 3 and 6) by { and }.
2. No restriction is imposed on the length of a sequence of metaletters in a metavariable. However, we have not bothered to include a mechanism to allow arbitrary length of a sequence of metadigits. At most five metadigits are permitted in an indexed metavariable.

List of flexowriter symbols:

a, b ..., z, A, B, ..., Z, 0, 1, ..., 9,

∧ ∨ × / = ; [ ] ( ) | < > " ' + ? : ¬ - . $_{10}$,

For the separation of underlined sequences of flexowriter symbols the lay-out symbols space, tab and new line carriage return are used.

```
begin   comment Definition of the processor,de Bakker,R1111,211066;

integer bound V,bound Sequence,bound Im,bound M,bound Commas,
        bound auxu,bound auxm,bound Metava,bound Underlined symbol;
        bound Im:= read; bound M:= read; bound V:= read;
        bound Sequence:= read; bound Commas:= read; bound auxu:= read;
        bound auxm:= read; bound Metava:= read;
        bound Underlined symbol:= read;

  begin
    integer space,tab,newline,blank,erase,bar,underlining,less,more,
            upper case,lower case,left par,right par,stopcode;
    integer im,in,is,va,co,tr,leftmetapar,rightmetapar,
            leftquote,rightquote,optopen,optclose,terminator;
    integer case,next RFS,next symbol,index;
    integer s,v,number of metavariables,number of underlined symbols,
            k,l,number of truths,c,m;
    boolean fit,first;
    integer array V[0:bound V],Sequence[0:bound Sequence],
            Metava[0:bound Metava],
            Underlined symbol[0:bound Underlined symbol],
            auxm[0:bound auxm],auxu[0:bound auxu],Im,Is,Comma,
            Length1,Length2[0:bound Im],Commas[0:bound Commas],
            M1,M2,M3,M4[0:bound M];

    comment V is the list of truths,Sequence the sequence that is
            evaluated.Metava,Underlined symbol,auxm and auxu are used
            for the representation of metavariables and underlined
            symbols by integers.Im,Is,Comma,Length1,Length2 are used
            for the administration of V.Commas is used for non-simple
            right parts.M1,M2,M3,M4 are used to store information
            about similarity of indexed metavariables;

    procedure init0 ;
    begin   comment Initialization of some global variables.The array
                    Underlined symbol is filled with the underlined
                    metasymbols,≤ and ≥ ;

            integer i,m,n,s,v,a,c,o,t,r;
            space    := read;   lower case   := read;   erase := read;
            tab      := read;   upper case   := read;   blank := read;
            newline  := read;   underlining  := read;   bar   := read;

            less     := read;   right par    := read;
            more     := read;   left par     := read;
            stopcode := read;
```

```
        im := 1;    va := 4;   leftmetapar  := 7;
        in := 2;    co := 5;   rightmetapar := 8;
        is := 3;    tr := 6;

        optopen  := 9;   leftquote  := -1;   terminator := -10;
        optclose :=10;   rightquote := -2;

        i:= read; m:= read; n:= read; s:= read; v:= read;
        a:= read; c:= read; o:= read; t:= read; r:= read;

        l:= number of underlined symbols:= 0;

        for Underlined symbol[l]:= 0,i,m,i,n,i,s,v,a,c,o,t,r,
                                   left par,right par,less,more
                                   do l:=l + 1;

        l:= 16; Underlined symbol[l]:= more;

        for auxu[number of underlined symbols]:= 0,2,4,6,8,10,12,
                                                  13,14,15,16 do
        number of underlined symbols:= number of underlined
                                               symbols + 1;
        number of underlined symbols:= 10;
        auxu[number of underlined symbols]:= 16
    end Init0;

procedure Init;
begin comment Initialization of the evaluation of a name;
      case:= next RFS:= next symbol:= s:= v:= number of metavariables:=
      number of truths:= c:= m:= auxm[0]:= 0;
      Comma[0]:= -1;
      first:= true;

      for k:= 0 step 1 until bound Im do
          Im[k]:= Is[k]:= Length2[k]:= 0;
      for k:= 0 step 1 until bound Commas do Commas[k]:= 0;

      k:= 0;

      number of underlined symbols:= 10; l:= 16;
      RFS(true); symbol
end   Init;
```

```
integer procedure RFS (f); value f; boolean f;
begin comment RFS reads a flexowriter symbol.The parameter f determines
              whether the symbols space,tab and newline are skipped;
      integer heptad; RFS:= next RFS;
              if next RFS = stopcode then goto end;
L:    heptad:= REHEP;
              if heptad = blank ∨ heptad = erase ∨
                 f ∧ (heptad = tab ∨ heptad = space ∨ heptad = newline)
              then goto L;
              if heptad = lower case then begin case:=   0; goto L end;
              if heptad = upper case then begin case:= 128; goto L end;
      next RFS:= heptad + (if heptad = stopcode ∨ heptad = space ∨
                           heptad = tab ∨ heptad = newline
                           then 0 else case);
end:
end RFS;

procedure symbol;
begin comment To the global variable next symbol an integer is
              assigned,representing:
              one of the symbols ⊀ or ⊁,or
              a metavariable,or
              an underlined terminal symbol,or
              an underlined metasymbol,or
              a non-underlined terminal symbol;
      integer temp; index:= 0;

start:temp:= RFS(true);
      if temp = bar
              then begin if next RFS = less then
                         begin RFS(true);
                               next symbol:= leftquote
                         end   else
                         if next RFS = more then
                         begin RFS(true);
                               next symbol:= rightquote
                         end   else
                         goto  terminal
                   end   else
      if temp = less
              then begin temp:= read metavariable(less,start,open);
                         RFS(true);
                         next symbol:= temp
                   end   else
```

```
    if temp = underlining
            then begin next symbol:= read underlined symbol;
                 open: if next symbol = optopen then
                       begin temp:= read metavariable(optopen,
                                        start,open);
                             next symbol:= temp + 200
                       end
                 end   else
    if temp = stopcode  then
            next symbol:= terminator else
terminal:
    next symbol:= temp + 300
end   symbol;

integer procedure read metavariable(f,start,open);
        value f; integer f; label start,open;
begin comment The metavariable is represented by an integer.
              Complications are caused by the possibility of the
              occurrence of sequences such as: < ab12 >.
              This is not a metavariable,but a sequence of six
              terminal symbols;
      integer i,j,k1,k2,aux,length;
      aux:= read metavariable:= 0;
      k1:= k;
      for i:= next RFS while metaletter(i) do
      begin RFS(true); k:= k + 1; Metava[k]:= i end;
      k2:= k;
      for i:= next RFS while metadigit(i)  do
      begin RFS(true); k:= k + 1; Metava[k]:= i end;
      if next RFS = underlining then
      aux:= read underlined symbol;
      if k1 = k2 ∨ (if f = less then next RFS ≠ more
                                else aux ≠ optclose) then
      begin s:= s + 1; Sequence[s]:= if f = less then less + 300
                                                  else optopen;
            for i:= k1 + 1 step 1 until k do
            begin s:= s + 1; Sequence[s]:= Metava[i] + 300 end;
            k:= k1;
            if aux ≠ 0 then
            begin next symbol:= aux; goto open end else
            goto start
      end;
      for i:= k2 + 1 step 1 until k do
      index:= index × 33 + Metava[i]; index:= index + 1000;
      k:= k2;
      length:= k − k1;
```

```
        for i:= 1 step 1 until number of metavariables do
        begin if auxm[i] - auxm[i - 1] = length then
              begin for j:= 1 step 1 until length do
                    if Metava[auxm[i - 1] + j] ≠
                       Metava[k1 + j] then goto out;
                    read metavariable:= i + 600 + (if
                                         index > 1000 then 100 else 0);
                    k:= k1; goto end
              end;
        out:
        end;  number of metavariables:= number of metavariables + 1;
              read metavariable:= number of metavariables + 600 +
                                 (if index > 1000 then 100 else 0);
              auxm[number of metavariables]:= k;
end:
end     read metavariable;

integer procedure read underlined symbol;
begin comment The underlined symbol is represented by an integer;
        integer temp,l1,i,j,length;
        boolean under; l1:= l; under:= true;
L:      if next RFS = underlining then
        begin under:= true; RFS(false); goto L end;
        if next RFS = space ∨ next RFS = tab ∨ next RFS = newline then
        begin if under then error(1) else RFS(true) end;
        if under then
        begin l:= l + 1; Underlined symbol[l]:= next RFS;
              under:= false; RFS(false); goto L
        end;
        if l1 = l then error(2);
        length:= l - l1;
        for i:= 1 step 1 until number of underlined symbols do
        begin if auxu[i] - auxu[i - 1] = length then
              begin for j:= 1 step 1 until length do
                    if Underlined symbol[auxu[i - 1] + j] ≠
                       Underlined symbol[l1 + j] then goto out;
                    read underlined symbol:= i; l:= l1; goto end
              end;
        out:
        end;  read underlined symbol:= number of underlined symbols:=
                    number of underlined symbols + 1;
              auxu[number of underlined symbols]:= l;
end:
end     read underlined symbol;
```

```
procedure output0(sw,e,f,g,h,A); value sw,e,f,g,h;
          integer sw,e,f,g,h; integer array A;
begin integer i,j,k,uj,vi,case;
      own integer N;
      switch switch := CO,IR,TS,SN,CV;
      procedure P(f); value f; integer f;
               if f = lower case then
               begin if case ≠ lower case then
                     begin case:= lower case; PUHEP(lower case) end
               end   else
               if f = upper case then
               begin if case ≠ upper case then
                     begin case:= upper case; PUHEP(upper case) end
               end   else PUHEP(f);

      procedure P1(f1,f2,f3,f4); value f1,f2,f3,f4;
                integer f1,f2,f3,f4;
      begin    P(space); P(f1); P(underlining); P(f2); P(f3); P(f4);
               P(space);
      end;

      procedure punch metadigits(f); value f; integer f;
      begin    integer a,j,k;
               integer array A[1 : 5];
               f:= f - 1000;
               k:= 0;
      L:       a:= f : 33 × 33; k:= k + 1; A[k]:= f - a;
               if a > 0 then begin f:= f : 33; goto L end;
               for j:= k step - 1 until 1 do
               begin P(lower case); P(A[j]) end
      end      punch metadigits;

      procedure punch metav(f); value f; integer f;
      begin    integer k; if f < 3 then
               begin P(lower case); P(less) end else
               begin P(lower case); P(underlining); P(less) end;
               for k:= 1 + auxm[vi - 501 - 100 × f] step 1 until
                          auxm[vi - 500 - 100 × f] do
               begin if Metava[k] > 128 then
                     begin P(upper case); P(Metava[k] - 128) end else
                     begin P(lower case); P(Metava[k]) end
               end;
               if f = 2 ∨ f = 4 then
               begin i:= i + 1; punch metadigits(A[i]) end;
               if f < 3 then
               begin P(upper case); P(more) end else
               begin P(lower case); P(underlining); P(upper case);
                     P(more); P(space)
               end
      end      punch metav;
```

```
procedure punch(f); value f; integer f;
begin   vi:=f;
        if vi = leftquote then
        P1(upper case,underlining,lower case,less) else
        if vi = rightquote then
        P1(upper case,underlining,underlining,more) else
        if vi = in then
        P1(lower case,i,underlining,n) else
        if vi = im then
        P1(lower case,i,underlining,m) else
        if vi = is then
        P1(lower case,i,underlining,s) else
        if vi = va then
        P1(lower case,v,underlining,a) else
        if vi = co then
        P1(lower case,c,underlining,o) else
        if vi = tr then
        P1(lower case,t,underlining,r) else
        if vi = leftmetapar then
        P1(lower case,underlining,upper case,left par) else
        if vi = rightmetapar then
        P1(lower case,underlining,upper case,right par) else
        if Terminal symbol(vi) then
        begin if vi > 300 ∧ vi < 428 then
              begin P(lower case); P(vi - 300) end else
              if vi > 428          then
              begin P(upper case); P(vi - 428) end else
              begin P(space);
                    for j:= 1 + auxu[vi - 1] step 1 until
                           auxu[vi] do
                    begin uj:= Underlined symbol[j];
                          if uj > 128 then
                          begin P(lower case); P(underlining);
                                P(upper case); P(uj - 128)
                          end   else
                          begin P(lower case); P(underlining);
                                P(uj)
                          end
                    end;  P(space)
              end
        end   else
        if Simple metav(vi) then punch metav(1) else
        if Ind simple metav(vi) then punch metav(2) else
        if Opt metav(vi) then punch metav(3) else
        if Ind opt metav(vi) then punch metav(4)
end     punch;
```

```
        procedure punch truth(j); value j; integer j;
        begin  PUNLCR; ABSFIXP(2,0,j); PUHEP(upper case);
               PUHEP(107); PUHEP(space); PUHEP(case);
               for i:= Comma[j - 1] + 2 step 1 until Comma[j] do
               punch(A[i]);
               if j < number of truths then punch(co)
        end    punch truth;


        procedure P2(string); string string;
        begin PUNLCR; PUTEXT(string);ABSFIXP(2,0,h);PUTEXT(⪡): ⪢);
              for i:= e step 1 until f do punch(A[i])
        end   P2;

        case:= 0; goto switch[sw];


CO:     P2(⪡CO(⪢); goto end;
IR:     P2(⪡IR(⪢); goto out;
TS:     P2(⪡TS(⪢); goto out;
SN:     PUNLCR;PUTEXT(⪡SN:  ⪢);
        for i:= e step 1 until f do punch(A[i]); PUNLCR; goto out;
CV:     PUNLCR;PUTEXT(⪡CV:⪢);PUNLCR;
        if first then
        begin for k:= 1 step 1 until number of truths do punch truth(k);
              first:= false; N:= number of truths
        end   else
        begin punch truth(1); PUNLCR; PUTEXT(⪡        .⪢); PUNLCR;
              PUTEXT(⪡        .⪢);PUNLCR; PUTEXT(⪡        .⪢);
              for k:= N step 1 until number of truths do punch truth(k)
        end;
        PUNLCR;
out:    if g ≠ 0 then
        begin P1(lower case,i,underlining,n);
              vi:= abs(g); punch metav(1)
        end;
end:
end     output0;
```

```
boolean procedure Terminal symbol(f); value f; integer f;
        Terminal symbol:= 8 < f ∧ f < 600;

boolean procedure Simple metav(f); value f; integer f;
        Simple metav:= 600 < f ∧ f < 700;

boolean procedure Metav(f); value f; integer f;
        Metav:= 600 < f ∧ f < 1000;

boolean procedure Ind metav(f); value f; integer f;
        Ind metav:= 700 < f ∧ f < 800 ∨ 900 < f ∧ f < 1000;

boolean procedure Opt metav(f); value f; integer f;
        Opt metav:= 800 < f ∧ f < 900;

boolean procedure Ind simple metav(f); value f; integer f;
        Ind simple metav:= 700 < f ∧ f < 800;

boolean procedure Ind opt metav(f); value f; integer f;
        Ind opt metav:= 900 < f ∧ f < 1000;

boolean procedure Non ind metav(f); value f; integer f;
        Non ind metav:= 600 < f ∧ f < 700 ∨ 800 < f ∧ f < 900;

comment The boolean procedures Terminal symbol(f),...,
        Non ind metav(f),are true,if the integer f represents
        a terminal symbol,...,a non indexed metavariable;

boolean procedure similar(f,g,h); value f,g,h; integer f,g,h;
        similar:= (M1[f] = g ∨ M1[f] = g + 200 ∨
                   M1[f] + 200 = g) ∧ M2[f] = h;

boolean procedure metadigit(f); value f; integer f;
        metadigit:= 0 < f ∧ f < 9 ∨ 18 < f ∧ f < 26 ∨ f = 32;

boolean procedure metaletter(f); value f; integer f;
begin   integer temp; temp:= if f > 128 then f - 128 else f;
        metaletter:=  34 < temp ∧ temp < 42 ∨ 49 < temp ∧ temp < 57 ∨
                      66 < temp ∧ temp < 74 ∨ 80 < temp ∧ temp < 89 ∨
                      96 < temp ∧ temp <105 ∨114 < temp ∧ temp <122
end     metaletter;
```

```
procedure error(f); value f; integer f;
begin PUNLCR; PUTEXT(⊀ error ⊁); ABSFIXP(3,0,f);
      goto end program
end   error;

procedure add to Sequence;
begin s:= s + 1; Sequence[s]:= next symbol;
      if index > 1000 then begin s:= s + 1; Sequence[s]:= index end;
      symbol
end   add to Sequence;

procedure add to V(A,f,g); value f,g; integer f,g; integer array A;
begin comment The value of a simple name is added to V.The
              administration of the arrays Im,Is,Comma,Length1,
              Length2 is updated;
      integer par,quote,k,sk;
      boolean comma,right of is;
      par:= quote:= 0; number of truths:= number of truths + 1;
      comma:= right of is:= false;
      if number of truths > 1 then begin v:= v + 1; V[v]:= co end;
      for k:= f step 1 until g do
      begin sk:= A[k];
             if Terminal symbol(sk) then goto add1;
             if Non ind metav(sk)   then goto add0;
             if Ind metav(sk)       then
                                    begin
                                      v:= v + 1; V[v]:= sk;
                                      k:= k + 1; sk:= A[k];
                                      if ¬(Ind opt metav(A[k - 1]) ∨
                                      right of is) then goto add2
                                    end else
             if sk = leftquote      then quote:= quote + 1 else
             if sk = rightquote     then quote:= quote - 1 else
             if sk = leftmetapar    then
                                    begin
                                      if par = 0 ∧ quote = 0 then
                                      comma:= true; par:= par + 1
                                    end else
             if sk = rightmetapar   then par:= par - 1 else
```

```
          if sk = im              then
                                  begin
                                    if quote = 0 then
                                    begin
                                      Im[number of truths]:= v;
                                      Length1[number of truths]:=
                                        Length2[number of truths];
                                      Length2[number of truths]:= 0
                                    end
                                  end else
          if sk = is              then
                                  begin
                                    if quote = 0 then
                                    begin
                                      Is[number of truths]:= v;
                                      right of is:= true
                                    end
                                  end else
          if sk = co              then
                                  begin
                                    if quote = 0 ∧ par = 0 then
                                    begin
                                      Comma[number of truths]:= v;
                                      number of truths:=
                                        number of truths + 1;
                                      right of is:= false
                                    end else
                                    if quote = 0 ∧ par = 1 then
                                    begin
                                      if comma then
                                      begin
                                        c:= c + 1;
                                        Commas[c]:= -number of truths;
                                        comma:= false
                                      end;
                                      c:= c + 1; Commas[c]:= v
                                    end
                                  end else
          if sk = in              then goto add1;
          goto add;
   add0:  if Opt metav(sk)        then goto add;
   add1:  if right of is          then goto add;
   add2:  Length2[number of truths]:= Length2[number of truths] + 1;
   add:   v:= v + 1; V[v].= sk
   end;   Comma[number of truths]:= v;
          output0(5,1,v,0,0,V)
end    add to V;
```

```
procedure NAME;
begin comment See the introduction at the beginning of this chapter;
      SIMPLE NAME;
      if Simple metav(Sequence[s]) then
      begin s:= s - 2;
            output0(4,1,s,-Sequence[s + 2],0,Sequence);
            evaluate(fit,1,s,s + 1,Sequence,- Sequence[s + 2])
      end else
      begin output0(4,1,s,0,0,Sequence);
            evaluate(fit,1,s,s + 1,Sequence,0)
      end;
      addto V(Sequence,1,s); s:= 0;
      if next symbol = co then
      begin symbol; NAME end else
      if next symbol ≠ terminator then error(3)
end   NAME;

comment The procedures SIMPLE NAME to SIMPLE METAVARIABLE test the
        syntax of a simple name,when it is read from the input tape.
        If the simple name contains a simple primary,this is evaluated
        in the procedure SIMPLE FACTOR;

procedure SIMPLE NAME;
      if next symbol = tr        then add to Sequence else
      if next symbol = leftquote then
                                 begin
                                   add to Sequence;
                                   LIST OF METAEXPRESSIONS;
                                   if next symbol = rightquote then
                                   add to Sequence else error(4)
                                 end else
      if Terminal symbol(next symbol) ∨
         next symbol = va        then SIMPLE TERM
                                 else error(5);

procedure LIST OF METAEXPRESSIONS;
begin METAEXPRESSION;
      if next symbol = co then
      begin add to Sequence;
            LIST OF METAEXPRESSIONS
      end
end   LIST OF METAEXPRESSIONS;

procedure SIMPLE TERM;
begin SIMPLE FACTOR;
      if next symbol = in then
      begin add to Sequence;
            SIMPLE METAVARIABLE
      end
end   SIMPLE TERM;
```

```
procedure SIMPLE FACTOR;
      if Terminal symbol(next symbol)
                                        then
                                        begin
                                          add to Sequence;
                                          SIMPLE FACTOR
                                        end else
      if next symbol = va               then
                                        begin
                                          symbol;
                                          if next symbol = leftmetapar then
                                          begin
                                            integer aux2;
                                            symbol; aux2:= s + 1;
                                            if Terminal symbol(next symbol)
                                            then TERMINAL SEQUENCE else
                                            error(6);
                                            evaluate(fit,aux2,s,s + 1,
                                                     Sequence,0);
                                            if next symbol = rightmetapar
                                            then symbol else error(7);
                                            SIMPLE FACTOR
                                          end else error(8)
                                        end;

procedure METAEXPRESSION;
      if next symbol = tr               then
                                        begin
                                          add to Sequence;
                                          if next symbol = im then
                                          begin
                                          L1: add to Sequence;
                                              if Terminal symbol(next symbol)V
                                                 Metav(next symbol) then
                                              begin
                                                 LEFTPART;
                                              L2: if next symbol = is then
                                                  begin
                                                  L3: add to Sequence;
                                                      RIGHTPART
                                                  end
                                              end else error(9)
                                          end else error(10)
                                        end else
```

```
        if Terminal symbol(next symbol) V
           Metav(next symbol)           then
                                        begin
                                          METASEQUENCE;
                                          if next symbol = im then
                                                     goto L1 else
                                          if next symbol = is then
                                                     goto L3 else
                                          if next symbol = in then
                                          begin
                                            add to Sequence;
                                            SIMPLE METAVARIABLE;
                                            goto L2
                                          end
                                        end
                                        else error(11);

procedure LEFTPART;
begin METASEQUENCE;
        if next symbol = in             then
                                        begin
                                          add to Sequence;
                                          SIMPLE METAVARIABLE
                                        end
end LEFTPART;

procedure RIGHTPART;
        if next symbol = leftmetapar
                                        then
                                        begin
                                          add to Sequence;
                                          LIST OF SIMPLE RIGHTPARTS;
                                          if next symbol = rightmetapar
                                          then add to Sequence else error(12)
                                        end
                                        else SIMPLE RIGHTPART;

procedure LIST OF SIMPLE RIGHTPARTS;
begin SIMPLE RIGHTPART;
        if next symbol = co             then
                                        begin
                                          add to Sequence;
                                          LIST OF SIMPLE RIGHTPARTS
                                        end
end LIST OF SIMPLE RIGHTPARTS;
```

```
procedure SIMPLE RIGHTPART;
      if next symbol = tr           then add to Sequence else
      if next symbol = leftquote    then
                                    begin
                                      add to Sequence;
                                      LIST OF METAEXPRESSIONS;
                                      if next symbol = rightquote then
                                      add to Sequence else error(13)
                                    end else
      if next symbol = va ∨
         Ind metav(next symbol) ∨
         Terminal symbol(next symbol)
                                    then IND METATERM else error(14);

procedure IND METATERM;
begin IND METAFACTOR;
      if next symbol = in then
      begin add to Sequence; SIMPLE METAVARIABLE end
end   IND METATERM;

procedure IND METAFACTOR;
      if Terminal symbol(next symbol) ∨
         Ind metav(next symbol)     then
                                    begin
                                      add to Sequence;
                                      IND METAFACTOR
                                    end else
      if next symbol = va           then
                                    begin
                                      add to Sequence;
                                      if next symbol = left metapar then
                                      begin
                                        add to Sequence;
                                        if Terminal symbol(next symbol) ∨
                                        Ind metav(next symbol) then
                                        IND METASEQUENCE else error(15);
                                        if next symbol = right metapar
                                        then add to Sequence
                                        else error(16);
                                        IND METAFACTOR
                                      end else error(17)
                                    end;

procedure TERMINAL SEQUENCE;
      if Terminal symbol(next symbol)
                                    then
                                    begin
                                      add to Sequence;
                                      TERMINAL SEQUENCE
                                    end TERMINAL SEQUENCE;
```

```
procedure METASEQUENCE;
      if Terminal symbol(next symbol) ∨
         Metav(next symbol)          then
                                     begin
                                       add to Sequence;
                                       METASEQUENCE
                                     end METASEQUENCE;

procedure IND METASEQUENCE;
      if Terminal symbol(next symbol) ∨
         Ind metav(next symbol)      then
                                     begin
                                       add to Sequence;
                                       IND METASEQUENCE
                                     end IND METASEQUENCE;

procedure SIMPLE METAVARIABLE;
      if Simple metav(next symbol)
                                     then add to Sequence else error(18);

boolean procedure envelope(l,a,b,c,A,B,p,q,para,n,n0);
        value l,a,b,c,p,q,para,n,n0;
        integer l,a,b,c,p,q,para,n,n0;
        integer array A,B;
begin comment  envelope is true,if the sequence in the array V,
               from V[p] to V[q],is an envelope of the sequence in the
               array A,from A[a] to A[b].Otherwise,envelope is false.
               The array A has as its corresponding actual either the
               array Sequence or the array V(the latter case occurs
               when it is tested whether a derived condition is an
               envelope of a truth in V).l is the length of the
               sequence V[p],...,V[q],decreased by the number of
               (indexed) optional metavariables in this sequence.
               c points to the first free place in the array A.
               This is used for auxiliary evaluations,e.g.of the value
               of a derived condition.A[a],...,A[b] contain the
               terminal sequence of a simple sequence.para(≠0)
               represents the simple metavariable of the simple
               sequence in case such a simple metavariable is present.
               B,n,n0 are used in the mechanism for testing whether
               subsequences,belonging to similar metavariables,are
               equal.envelope is defined recursively:
               V[p],...,V[q] is an envelope of A[a],...,A[b],if V[p]
               and an appropriate initial sequence of A[a],...,A[b]
               fulfil the requirements of 2.2.3,step 3a,and
               V[p + 1],...,V[q] is an envelope of the remaining
               part of A[a],...,A[b];
```

```
integer Vp,temp;
boolean index,opt,fit;

integer procedure next l;
   next l:= l - (if opt then 0 else 1);

integer procedure next p;
   next p:= p + (if index then 2 else 1);

boolean procedure last;
   last:= p + (if index then 1 else 0) = q;

integer procedure reduced Vp;
   reduced Vp:= Vp - (if opt ∧ index then 300 else
                 if opt then 200 else if index then 100 else 0);

procedure add to M(f,g); value f,g; integer f,g;
   if index then
   begin m:= m + 1; M1[m]:=Vp; M2[m]:= V[p + 1];
         M3[m]:= f; M4[m]:= g
   end add to M;

boolean procedure env(a); value a; integer a;
   env:= envelope(next l,a,b,c,A,B,next p,q,0,n,n0);

boolean procedure TERMINAL;
   TERMINAL:= if Vp = A[a] then (if last then a = b else
                                 env(a + 1)) else false;

boolean procedure SIMP MET;
begin
   SIMP MET:= false;
   if last         then evaluate(fit,a,b,c,A,reduced Vp) else
                   begin temp:= temp + 1;
                         if l > b - temp + 1 then goto end;
                         evaluate(fit,a,temp,c,A,reduced Vp)
                   end;
   if fit ∧ last   then
                   begin add to M(a,b); SIMP MET:= true end else
```

```
   if fit           then
                    begin add to M(a,temp);
                      if env(temp + 1) then
                      SIMP MET:= true else
                      begin if index then m:= m - 1;
                        SIMP MET:= SIMP MET
                      end
                    end else
   if ¬ last        then SIMP MET:= SIMP MET;
end:
end SIMP MET;

boolean procedure OPT MET;
begin OPT MET:= false; opt:= true;
   if last          then
                    begin if a > b then
                      begin OPT MET:= true; add to M(0,-1) end
                      else OPT MET:= SIMP MET
                    end
                    else
                    begin add to M(0,-1);
                      if env(a) then OPT MET:= true else
                      begin if index then m:= m - 1;
                        if 1 ≤ b - a then OPT MET:= SIMP MET
                      end
                    end
end OPT MET;

boolean procedure IND SIMP MET;
begin index:= true; IND SIMP MET:= SIMILAR end IND SIMP MET;

boolean procedure IND OPT MET;
begin index:= opt:= true; IND OPT MET:= SIMILAR end IND OPT MET;

boolean procedure SIMILAR;
begin integer i1,i2,temp1,temp2;
   SIMILAR:= false;
   for i1:= n + 1 step 1 until m do
   if similar(i1,Vp,V[p + 1]) then
   begin temp1:= M3[i1]; temp2:= M4[i1];
     if b - a - temp2 + temp1 < next 1 then goto end;
     for i2:= temp1 step 1 until temp2 do
     if A[a + i2 - temp1] ≠ (if i1 > n0 ∧ n0 > 0 then
     V[i2] else B[i2]) then goto end;
```

```
        SIMILAR:= if last then (if temp2 > 0 then
        a + temp 2 - temp1 = b else a > b)
        else env(a + temp2 - temp1 + 1);
        goto end
      end;
      SIMILAR:= if opt then OPT MET else SIMP MET;
    end:
    end SIMILAR;

    envelope:= false;
    if para ≠ 0 then
                begin if abs(para) ≠ V[q] ∨ in ≠ V[q - 1]
                  then goto end else
                  begin q:= q - 2; l:= l - 2 end
                end;
    if l > b - a +1 then goto end;
    if a ≤ b ∧ ¬ Terminal symbol(A[a]) ∨ q ≥ p + 2 ∧ V[q - 1] = in
                 then goto end;
    opt:= index:= false; temp:= a - 1; Vp:= V[p];
    envelope:=  if Terminal symbol(Vp) then TERMINAL else
                if Simple metav(Vp)    then SIMP MET else
                if Opt metav(Vp)       then OPT MET  else
                if Ind simple metav(Vp)then IND SIMP MET else
                if Ind opt metav(Vp)   then IND OPT MET  else false;
end:
end envelope;

procedure evaluate(fi,a,b,c,A,para); value a,c,para;
      integer a,b,c,para; boolean fi; integer array A;
begin comment The value of the sequence A[a],...,A[b] is determined.
              c and para have the same meaning as in envelope.
              fi is used to store the result of auxiliary calls
              of evaluate in the body of envelope.
              error 19 occurs when the empty sequence is evaluated,
              and error 20 when the sequence is not simple;
      integer i,i1,temp1,temp2,temp3,temp4,n,n0,d,e,par;
      boolean metav,condition present,rightpart present;

      procedure tr1;
        if A[a] = tr ∧ a = b then
        begin fi:= true; goto end evaluate end;

      procedure metastring;
        if A[a] = leftquote then
        begin b:= b - 2;
          for i:= a step 1 until b do A[i]:= A[i + 1];
          goto end evaluate
        end;
```

```
procedure apply V;
begin  for i:= a step 1 until b do
       if ¬ Terminal symbol(A[i]) then error(20);
       for i:= number of truths step - 1 until 1 do
       begin consider truth(i);
             if envelope(Length2[i],a,b,if para > 0 then
             c else b + 1,A,A,if condition present then
             temp2 + 2 else temp1,if rightpart present then
             temp3 else temp4,para,n,n0) then
             begin if condition satisfied then
                   evaluate right part
             end
       end
end    apply V,

procedure consider truth(i); value i; integer i;
begin  temp1:= Comma[i - 1] + 2; temp2:= Im[i];
       temp3:= Is[i]; temp4:= Comma[i]; m:= n;
       condition present:= temp2 ≠ 0;
       right part present:= temp3 ≠ 0
end    consider truth;

boolean procedure condition satisfied;
begin  condition satisfied:= true;
       if condition present then
       begin derive condition(metav,A,temp1,temp2,c,d,n);
             if ¬ metav then
                        begin
                          output0(1,c,d,0,i,Sequence);
                          evaluate(fi,c,d,d + 1,Sequence,0);
                          condition satisfied:= fi
                        end
                        else
                        begin n0:= m;
                          for i1:= number of truths step -1
                                                     until 1 do
                          begin
                            if Im[i1] ≠ 0 ∨ Is[i1] ≠ 0 then
                            goto end i1;
                            if envelope(Length1[i],
                                   Comma[i1 - 1] + 2,Comma[i1],
                                   c,V,A,temp1,temp2,0,n,n0)
                            then goto end;
                            end i1:
                          end;
                          n0:= - 1; condition satisfied:= false
                        end
       end;
end:
end    condition satisfied;
```

```
        procedure evaluate right part;
        begin  if ¬ right part present
                                    then
                                    begin fi:= true;
                                      if para ≤ 0 then
                                      begin b:= a; A[a]:= tr end
                                    end else
                if para > 0         then
                                    begin
                                      derive right part(i,a,b,c,e,A,
                                            temp3 + 2,temp4,par,n,n0);
                                      evaluate(fi,c,e,e + 1,Sequence,- par)
                                    end else
                                    begin
                                      derive right part(i,a,b,a,b,A,
                                            temp3 + 2,temp4,par,n,n0);
                                      output0(2,a,b,par,i,A);
                                      evaluate(fi,a,b,b + 1,Sequence,- par)
                                    end;
                goto end evaluate
        end     evaluate right part;

        if a > b then error(19);
        n:= m; n0:= - 1; fi:= false;
        tr1; metastring; apply V;
        if para < 0 then
        begin b:= b + 2; A[b - 1]:= in; A[b]:= - para end;
    end evaluate: m:= n
    end evaluate;

    procedure derive condition(fi,A,p,q,t,r,n);
              value p,t,q,n; integer p,q,r,t,n;
              boolean fi; integer array A;
    begin comment From the condition V[p],...,V[q] the derived condition
                  A[t],...,A[r] is derived.
                  fi is true if the condition contains a metavariable
                  which is similar to no indexed metavariable in the
                  left part concerned.
                  Information about similarity of metavariables is kept
                  in the arrays M1 to M4.n is a pointer of these arrays;
          integer i1,i2,i3,vi;
          procedure add to Seq(f); value f; integer f;
          begin r:= r + 1; Sequence[r]:= f end add to Seq;
          r:= t - 1; fi:= false;
```

```
          for i1:= p step 1 until q do
          begin  vi:= V[i1];
                 if ¬ Ind metav(vi) then
                 begin add to Seq(vi);
                   if Non ind metav(vi) then
                   fi:= true
                 end else
                 begin for i2:= n + 1 step 1 until m do
                   if similar(i2,vi,V[i1 + 1]) then
                   begin for i3:= M3[i2] step 1 until M4[i2] do
                     add to Seq(A[i3]); i1:= i1 + 1; goto out
                   end;
                   fi:= true; add to Seq(vi);
                   i1:= i1 + 1; add to Seq(V[i1]);
                 out:
                 end
          end
end       derive condition;

procedure derive rightpart(k,a,b,t,s,A,v1,v2,par,n,n0);
              value k,a,b,t,v1,v2,n,n0;
              integer k,a,b,t,s,v1,v2,par,n,n0;
              integer array A;
begin comment  From the right part V[v1],...,V[v2] the derived
               right part is constructed and the simple evaluation
               of the derived right part is performed,i.e.the
               derived simple right parts,except the last one,
               are evaluated and their values are added to V.
               The array aux is used for the temporary storage of
               the sequence that is evaluated,i.e.of A[a],...,A[b].
               k is the number of the truth that is applied.
               t,s,par,n,n0 are passed on to derive simple
               right part;
          integer k1;
          integer array aux[a:b];
          for k1:= a step 1 until b do aux[k1]:= A[k1];
          if V[v1] = left metapar then
          begin  integer p,q,aux1,aux2;
                 aux1:= v1 + 1;
                 for p:= 1 step 1 until c do
                 if Commas[p] = - k then
                 begin q:= p + 1; aux2:= Commas[q]; goto L end;
                 derive simple rightpart(k,t,s,aux,v1 + 1,v2 - 1,par,n,n0);
                 goto out;
          L:     derive simple rightpart(k,t,s,aux,aux1,aux2,par,n,n0);
                 output0(2,t,s,par,k,Sequence);
                 evaluate(fit,t,s,s + 1,Sequence,-par);
                 add to V( Sequence,t,s);
```

```
            if Commas[q + 1] > 0 then
            begin aux1:= aux2 + 2; q:= q + 1;
                  aux2:=Commas[q]; goto L
            end else
            derive simple rightpart(k, t,s,aux,aux2 + 2,
                                        v2 - 1,par,n,n0);
      out:
      end   else
            derive simple rightpart(k,t,s,aux,v1,v2,par,n,n0);
            m:= n
end   derive right part;

procedure derive simple rightpart(k,t,s,aux,v1,v2,par,n,n0);
          value k,t,v1,v2,n,n0;
          integer k,t,s,v1,v2,par,n,n0;
          integer array aux;
begin comment  From the simple right part V[v1],...,V[v2] the
               derived simple right part Sequence[t],...,Sequence[s]
               is constructed.
               The array aux was used in derive right part
               for temporary storage of the sequence that is
               evaluated.If the simple right part is a simple term
               which contains the metasymbol in,then par is used to
               store the simple metavariable of this simple term.
               n,n0 are used for the administration of similarity
               of indexed metavariables.If the derived simple
               right part contains a simple primary,then this
               simple primary is replaced by the value of its
               terminal sequence;
      integer i1,i2,i3,temp,vi1,quote;
      boolean val;
      procedure add to Seq(f); value f; integer f;
      begin s:= s + 1; Sequence[s]:= f end add to Seq;
      s:= t - 1; val:= false; par:= quote:= 0;
      for i1:= v1 step 1 until v2 do
      begin  vi1:= V[i1];
             if vi1 = leftquote           then
                                          begin quote:= quote + 1;
                                            add to Seq(vi1)
                                          end else
             if vi1 = rightquote          then
                                          begin quote:= quote - 1;
                                            add to Seq(vi1)
                                          end else
             if vi1 = va ∧ quote = 0      then val:= true else
             if vi1 = leftmetapar ∧ val then temp:= s + 1 else
```

```
          if vi1 = rightmetapar ∧ va1then
                                     begin va1:= false;
                                       output0(3,temp,s,0,k,
                                              Sequence);
                                       evaluate( fit,temp,s,s + 1,
                                                Sequence,0)
                                     end else
          if vi1 = in ∧ quote = 0    then
                                     begin i1:= i1 + 1;
                                       par:= V[i1]
                                     end else
          if Ind metav(vi1)          then
                                     begin
                                       for i2:= n + 1 step 1 until
                                                               m do
                                       if similar(i2,vi1,V[i1 + 1])
                                       then
                                       begin
                                         for i3:= M3[i2] step 1
                                             until M4[i2] do
                                         add to Seq(if i2 > n0 ∧
                                             n0 > 0 then V[i3] else
                                             aux[i3]);
                                         i1:= i1 + 1; goto out
                                       end;
                                       add to Seq(vi1);
                                       i1:= i1 + 1;
                                       add to Seq(V[i1]);
                                     out:
                                     end else add to Seq(vi1)
     end
end  derive simple rightpart;

RUNOUT; PUNLCR; PUTEXT(⸨ results de Bakker,R1111,211066⸩);

Init0;
L: PUNLCR; PUNLCR; PUNLCR; PUNLCR; Init; NAME; goto L;

end program:
end
end
```

4.2. Examples

In this section we give fourteen examples of the evaluation of a name by the processor.

Section 4.2.1 contains some introductory examples, section 4.2.2 examples related to chapter 3, section 4.2.3 examples related to the definition of ALGOL 60, and section 4.2.4 Wang's algorithm for the propositional calculus.

We have tried to make these examples better comprehensible by the inclusion of some intermediate results.

The structure of each example is as follows:

a. Input.

The name which is to be evaluated is exhibited.

b. Output.

1. The successive simple names which constitute this name are given.
2. The contents of V are shown after the addition of the values of each of these simple names to V.
3. If a truth is applied then the number of this truth and the corresponding derived right part are exhibited.
4. The truths are numbered in the order which is the reverse of the order in which they are applied (2.3.5). For the sake of easier readability, we have supplied each truth in the output with its number. Occasionally, we omit a part of the contents of V, when this part has already been shown.
5. If a derived condition is a terminal sequence, then it is exhibited.
6. If a derived right part contains one or more simple primaries, then the terminal sequences of these simple primaries (i.e. the terminal sequences occurring after the va symbol) are shown separately, and the number of the corresponding truth is given.

The examples were run on the EL X8. They are printed directly from the output tape, except for the manual addition of some spaces and new line carriage return symbol. The time used for the fourteen examples was 31.5 minutes.

List of abbreviations:

SN : simple name,

CV : contents of V,

IR(i): intermediate result, i.e. derived (simple) right part, found by applying truth i,

TS(i): terminal sequence of simple primary, occurring in the derived right part of truth i,

CO(i): derived condition of truth i.

4.2.1. Introductory examples.

4.2.1.1. Example 1.

Greatest common divisor of two positive integers
by the Euclidean algorithm.
This example has already been treated in 2.4.2.1.

Input :

```
≮ 1 <integer> in <integer> co

  (<integer1>,<integer1><integer2>)  is  (<integer1>,<integer2>) co
  (<integer1><integer2>,<integer2>)  is  (<integer1>,<integer2>) co

  (<integer1>,<integer1>)  is  <integer1> ≯ co

  (11,111) co (1111,11)
```

Output :

```
 results de Bakker,R1111,211066


SN:
 ≮ 1<integer>  in <integer> co (<integer1>,<integer1><integer2>) is
 (<integer1>,<integer2>) co (<integer1><integer2>,<integer2>) is
 (<integer1>,<integer2>) co (<integer1>,<integer1>) is <integer1> ≯

CV:

  1 : 1<integer>  in <integer> co
  2 : (<integer1>,<integer1><integer2>) is (<integer1>,<integer2>) co
  3 : (<integer1><integer2>,<integer2>) is (<integer1>,<integer2>) co
  4 : (<integer1>,<integer1>) is <integer1>

SN:  (11,111)

IR(  2 ):  (11,1)
IR(  3 ):  (1,1)
IR(  4 ):  1
```

CV:

```
  1 : 1<integer> in <integer> co
        .
        .
        .
  4 : (<integer1>,<integer1>) is <integer1> co
  5 : 1
```

SN: (1111,11)

IR(3): (11,11)
IR(4): 11
CV:

```
  1 : 1<integer> in <integer> co
        .
        .
        .
  4 : (<integer1>,<integer1>) is <integer1> co
  5 : 1 co
  6 : 11
```

Remark:
One should realize that a subsequent evaluation of e.g. (1111,111) will result in the value tr by application of truth 5.If one considers this result undesirable,one may avoid it by changing truth 4 into (<integer1>,<integer1>) is ⊀ <integer1> ⊁ .
This is an example of a more general situation: If a metaprogram is applied to the evaluation of more than one simple sequence,one will have to take into account that the value of some simple sequence may be influenced by a previously added truth. Therefore,if we say that a metaprogram has a certain meaning,this is in general restricted to the case that only one simple sequence is evaluated.We may add to this,on the one hand,that it is often very useful to be able to influence subsequent evaluations (see e.g. examples 12 or 13),and on the other hand that it is often possible to avoid such effects,by taking some special measures,of which we have given an example above.

4.2.1.2. Example 2.

Lexicographical ordering. This example was treated in 2.4.2.2.

Input :

```
⊀ a  in  <letter> co
  b  in  <letter> co
  c  in  <letter> co
  d  in  <letter> co
  e  in  <letter> co

  <letter><word>  in  <word> co

  <word> pre <word>  is  false co

  <letter1> pre <letter2> im <letter1><word>  pre <letter2><word> co

  <letter1><word1> pre <letter1><word2>  is  <word1> pre <word2> co

  <letter1><word>  pre <letter1>  is  false  co

  <letter1>  pre  <letter1><word>  co

  <letter2>  pre  <letter3>  im
  <letter1>  pre  <letter3>  is  <letter1>  pre  <letter2> co

  <letter>  pre  a  is  false  co

  a  pre  b  co
  b  pre  c  co
  c  pre  d  co
  d  pre  e  co

  <letter1>  pre  <letter1> ⊁ co

  dbc  pre  dee  co  bca  pre  bb
```

Output :

SN:

⊦ a in <letter> co b in <letter> co c in <letter> co d in <letter> co e in <letter> co <letter><word> in <word> co <word> pre <word> is false co <letter1> pre <letter2> im <letter1><word> pre <letter2> <word> co <letter1><word1> pre <letter1><word2> is <word1> pre <word2> co <letter1><word> pre <letter1> is false co <letter1> pre <letter1><word> co <letter2> pre <letter3> im <letter1> pre <letter3> is <letter1> pre <letter2> co <letter> pre a is false co a pre b co b pre c co c pre d co d pre e co <letter1> pre <letter1> ⊣

CV:

1 : a in <letter> co
2 : b in <letter> co
3 : c in <letter> co
4 : d in <letter> co
5 : e in <letter> co
6 : <letter><word> in <word> co
7 : <word> pre <word> is false co
8 : <letter1> pre <letter2> im
<letter1><word> pre <letter2><word> co
9 : <letter1><word1> pre <letter1><word2> is <word1> pre <word2> co
10 : <letter1><word> pre <letter1> is false co
11 : <letter1> pre <letter1><word> co
12 : <letter2> pre <letter3> im
<letter1> pre <letter3> is <letter1> pre <letter2> co
13 : <letter> pre a is false co
14 : a pre b co
15 : b pre c co
16 : c pre d co
17 : d pre e co
18 : <letter1> pre <letter1>

SN: dbc pre dee

IR(9): bc pre ee
CO(8): b pre e
IR(12): b pre d
IR(12): b pre c
CV:

1 : a in <letter> co

.

.

.

18 : <letter1> pre <letter1> co
19 : tr

```
SN:  bca pre bb

IR(  9 ):  ca pre b
CO(  8 ):  c pre b
IR( 12 ):  c pre a
IR( 13 ):   false
IR(  7 ):   false
CV:

  1 : a in <letter> co
        .
        .
        .
 18 : <letter1> pre <letter1> co
 19 :  tr  co
 20 :  false
```

4.2.1.3.Example 3.

Definition of a row.

A row is defined as a sequence of letters,none of which are equal.This example is taken from [41],p. 17.

Input :

```
⊀ a  in  <letter> co
  b  in  <letter> co
  c  in  <letter> co

  <letter1>  el  <row1>  im  <letter1>  el  <row1><letter> co
  <letter1>  el  <row><letter1>  co
  <letter1>  el  <letter1>  co

  <row><letter>  in  <row>  co
  <letter1>  el  <row1> im <row1><letter1> in <row>  is  false  co
  <letter>  in  <row>  ⊁ co

  abc  in  <row>  co  abca  in  <row>
```

Output :

```
SN:
⊀ a in <letter> co b in <letter> co c in <letter> co <letter1> el
<row1> im <letter1> el <row1><letter> co <letter1> el <row><letter1>
co <letter1> el <letter1> co <row><letter> in <row> co <letter1> el
<row1> im <row1><letter1> in <row> is  false  co <letter> in <row> ⊁
```

```
CV:

  1 : a in <letter> co
  2 : b in <letter> co
  3 : c in <letter> co
  4 : <letter1> el <row1> im <letter1> el <row1><letter> co
  5 : <letter1> el <row><letter1> co
  6 : <letter1> el <letter1> co
  7 : <row><letter> in <row> co
  8 : <letter1> el <row1> im <row1><letter1> in <row> is  false  co
  9 : <letter> in <row>

SN:  abc in <row>

CO( 8 ):  b el a
CO( 8 ):  c el ab
CO( 4 ):  c el a
CO( 8 ):  b el a
CV:

  1 : a in <letter> co
        .
        .
        .
  9 : <letter> in <row> co
 10 :  tr

SN:  abca in <row>

CO( 8 ):  b el a
CO( 8 ):  b el a
CO( 8 ):  c el ab
CO( 4 ):  c el a
CO( 8 ):  b el a
CO( 8 ):  a el abc
CO( 8 ):  b el a
CO( 8 ):  b el a
CO( 4 ):  a el ab
CO( 4 ):  a el a
IR( 8 ):   false
CV:

  1 : a in <letter> co
        .
        .
        .
  9 : <letter> in <row> co
 10 :  tr  co
 11 :  false
```

4.2.1.4. Example 4.

Intermediate addition of truths to V.
This example shows how the first evaluation of the simple name a influences the second evaluation of a. This effect of intermediate addition of truths to V will be used extensively in the definition of ALGOL 60 in chapter 5.
Cf. also example 13.

Input :

```
⊀ a  in  <id> co
  b  in  <id> co

  a <id>  in  <id> co
  b <id>  in  <id> co

  <id1> is
    ( ⊀ <id2> is
      ( ⊀ <id3> is <id1>0<id2>0<id3>0 ⊁ co <id2>b ) ⊁ co
      <id1>b ) ⊁ co

  a co a
```

Output :

```
SN:
 ⊀ a in <id> co b in <id> co a<id> in <id> co b<id> in <id> co
 <id1> is  ( ⊀ <id2> is  ( ⊀ <id3> is <id1>0<id2>0<id3>0 ⊁ co
 <id2>b )  ⊁  co <id1>b ) ⊁

CV:

  1 : a in <id> co
  2 : b in <id> co
  3 : a<id> in <id> co
  4 : b<id> in <id> co
  5 : <id1> is  (  ⊀ <id2> is  (  ⊀ <id3> is <id1>0<id2>0<id3>0 ⊁  co
      <id2>b )  ⊁  co <id1>b )

SN:  a

IR( 5 ):   ⊀ <id2> is  (  ⊀ <id3> is a0<id2>0<id3>0 ⊁  co <id2>b ) ⊁
```

```
CV:

  1 : a in <id> co
       .
       .
       .
  5 : <id1> is  (  { <id2> is  (  { <id3> is <id1>0<id2>0<id3>0 }  co
      <id2>b )  }  co <id1>b )  co
  6 : <id2> is  (  { <id3> is a0<id2>0<id3>0 }  co <id2>b )

IR( 5 ):  ab
IR( 6 ):   { <id3> is a0ab0<id3>0 }
CV:

  1 : a in <id> co
       .
       .
       .
  5 : <id1> is  (  { <id2> is  (  { <id3> is <id1>0<id2>0<id3>0 }  co
      <id2>b )  }  co <id1>b )  co
  6 : <id2> is  (  { <id3> is a0<id2>0<id3>0 }  co <id2>b )  co
  7 : <id3> is a0ab0<id3>0

IR( 6 ):  abb
IR( 7 ):  a0ab0abb0
CV:

  1 : a in <id> co
       .
       .
       .
  5 : <id1> is  (  { <id2> is  (  { <id3> is <id1>0<id2>0<id3>0 }  co
      <id2>b )  }  co <id1>b )  co
  6 : <id2> is  (  { <id3> is a0<id2>0<id3>0 }  co <id2>b )  co
  7 : <id3> is a0ab0<id3>0 co
  8 : a0ab0abb0

SN:  a

IR( 7 ):  a0ab0a0
CV:

  1 : a in <id> co
       .
       .
       .
  5 : <id1> is  (  { <id2> is  (  { <id3> is <id1>0<id2>0<id3>0 }  co
      <id2>b )  }  co <id1>b )  co
  6 : <id2> is  (  { <id3> is a0<id2>0<id3>0 }  co <id2>b )  co
  7 : <id3> is a0ab0<id3>0 co
  8 : a0ab0abb0 co
  9 : a0ab0a0
```

4.2.2. Examples related to chapter 3.

4.2.2.1. Example 5.

Markov's algorithm for the greatest common divisor.
The construction of theorem 3.1.1 has been applied to the Markov algorithm for the g.c.d. which is defined in [33], p. 105.
(The extra symbol α is not necessary here.)

Input :

```
ⱶ 1  in  <symbol> co
  :  in  <symbol> co
  a  in  <symbol> co
  b  in  <symbol> co
  c  in  <symbol> co

  <symbol><tape>  in  <tape> co

  <tape1> :    <tape2>  is  <tape1>     <tape2> co
  <tape1> c    <tape2>  is  <tape1> 1   <tape2> co
  <tape1> a    <tape2>  is  <tape1> c   <tape2> co
  <tape1> b    <tape2>  is  <tape1> 1   <tape2> co
  <tape1> 1:   <tape2>  is  <tape1> :b  <tape2> co
  <tape1> 1:1  <tape2>  is  <tape1> a:  <tape2> co
  <tape1> 1a   <tape2>  is  <tape1> a1  <tape2> ⊣ co

  11:111
```

Output :

```
SN:
 ⱶ 1 in <symbol> co : in <symbol> co a in <symbol> co b in <symbol> co
 c in <symbol> co <symbol><tape>  in <tape> co <tape1> :<tape2>  is
 <tape1> <tape2>  co <tape1> c<tape2>  is <tape1> 1<tape2>  co
 <tape1> a<tape2>  is <tape1> c<tape2>  co <tape1> b<tape2>  is
 <tape1> 1<tape2>  co <tape1> 1:<tape2>  is <tape1> :b<tape2>  co
 <tape1> 1:1<tape2>  is <tape1> a:<tape2>  co <tape1> 1a<tape2>  is
 <tape1> a1<tape2>  ⊣
```

```
CV:

  1 : 1 in <symbol> co
  2 : : in <symbol> co
  3 : a in <symbol> co
  4 : b in <symbol> co
  5 : c in <symbol> co
  6 : <symbol><tape> in <tape> co
  7 : <tape1> :<tape2> is <tape1> <tape2> co
  8 : <tape1> c<tape2> is <tape1> 1<tape2> co
  9 : <tape1> a<tape2> is <tape1> c<tape2> co
 10 : <tape1> b<tape2> is <tape1> 1<tape2> co
 11 : <tape1> 1:<tape2> is <tape1> :b<tape2> co
 12 : <tape1> 1:1<tape2> is <tape1> a:<tape2> co
 13 : <tape1> 1a<tape2> is <tape1> a1<tape2>

SN:  11:111

IR( 12 ):  1a:11
IR( 13 ):  a1:11
IR( 12 ):  aa:1
IR(  9 ):  ca:1
IR(  9 ):  cc:1
IR(  8 ):  1c:1
IR(  8 ):  11:1
IR( 12 ):  1a:
IR( 13 ):  a1:
IR( 11 ):  a:b
IR( 10 ):  a:1
IR(  9 ):  c:1
IR(  8 ):  1:1
IR( 12 ):  a:
IR(  9 ):  c:
IR(  8 ):  1:
IR( 11 ):  :b
IR( 10 ):  :1
IR(  7 ):  1
CV:

  1 : 1 in <symbol> co
      .
      .
      .
 13 : <tape1> 1a<tape2> is <tape1> a1<tape2> co
 14 : 1
```

4.2.2.2. Example 6.

A Turing machine for addition.
The construction of theorem 3.1.2 has been applied to the Turing machine for addition which is defined in [16], p.12.

Input :

```
⊀ 0  in  <symbol> co
  1  in  <symbol> co

  <symbol><tape>  in  <tape> co

  q <state>  in  <state> co

  <state1><symbol1><symbol2><state2>  im
  <tape1><state1><symbol1><tape2>  is
  <tape1><state2><symbol2><tape2>  co

  <state1><symbol1>R<state2>  im
  <tape1><state1><symbol1><tape2>  is
  <tape1><symbol1><state2><tape2>  co

  <state1><symbol1>R <state2>  im
  <tape1><state1><symbol1>   is
  <tape1><symbol1><state2>0  co

  <state1><symbol1>L<state2>  im
  <tape1><symbol2><state1><symbol1><tape2>  is
  <tape1><state2><symbol2><symbol1><tape2>  co

  <state1><symbol1>L<state2>  im
  <state1>    <symbol1><tape1>  is
  <state2> 0 <symbol1><tape1>  co

  q   1 0 q   co
  q   0 R qq  co
  qq  1 R qq  co
  qq  0 R qqq co
  qqq 1 0 qqq ⊁ co

  q 1 1 0 1
```

Output :

SN:

⊀ 0 in <symbol> co 1 in <symbol> co <symbol><tape> in <tape> co
q<state> in <state> co <state1><symbol1><symbol2><state2> im <tape1>
<state1><symbol1><tape2> is <tape1> <state2><symbol2><tape2> co
<state1><symbol1>R<state2> im <tape1> <state1><symbol1><tape2> is
<tape1> <symbol1><state2><tape2> co <state1><symbol1>R<state2> im
<tape1> <state1><symbol1> is <tape1> <symbol1><state2>0 co <state1>
<symbol1>L<state2> im <tape1> <symbol2><state1><symbol1><tape2> is
<tape1> <state2><symbol2><symbol1><tape2> co <state1><symbol1>L
<state2> im <state1><symbol1><tape1> is <state2>0<symbol1><tape1>
co q10q co q0Rqq co qq1Rqq co qq0Rqqq co qqq10qqq ⊁

CV:

1 : 0 in <symbol> co
2 : 1 in <symbol> co
3 : <symbol><tape> in <tape> co
4 : q<state> in <state> co
5 : <state1><symbol1><symbol2><state2> im
<tape1> <state1><symbol1><tape2> is
<tape1> <state2><symbol2><tape2> co
6 : <state1><symbol1>R<state2> im
<tape1> <state1><symbol1><tape2> is
<tape1> <symbol1><state2><tape2> co
7 : <state1><symbol1>R<state2> im
<tape1> <state1><symbol1> is
<tape1> <symbol1><state2>0 co
8 : <state1><symbol1>L<state2> im
<tape1> <symbol2><state1><symbol1><tape2> is
<tape1> <state2><symbol2><symbol1><tape2> co
9 : <state1><symbol1>L<state2> im
<state1><symbol1><tape1> is
<state2>0<symbol1><tape1> co
10 : q10q co
11 : q0Rqq co
12 : qq1Rqq co
13 : qq0Rqqq co
14 : qqq10qqq

```
SN:  q1101

IR( 5 ):  q0101
IR( 6 ):  0qq101
IR( 6 ):  01qq01
IR( 6 ):  010qqq1
IR( 5 ):  010qqq0
CV:

  1 : 0 in <symbol> co
        .
        .
        .
 14 : qqq10qqq co
 15 : 010qqq0
```

4.2.2.3. Example 7.

Recognizer for the context sensitive language $\{a^n\ b^n\ a^n \mid n \geq 1\}$.

This example was treated in 3.2.3.

<u>Input</u> :

```
≮ a <as>   in  <as> co
  b <bs>   in  <bs> co

  aba  in  <ABA> co

  <as1>a<bs1>b<as1>a  in  <ABA>  is
  <as1> <bs1> <as1>   in  <ABA>  ≯ co

  aaabbbaaa  in  <ABA> co aaabbaaa  in  <ABA>
```

Output :

```
SN:
 ⊦ a<as>  in <as> co b<bs>  in <bs> co aba in <ABA> co
 <as1>a<bs1>b<as1>a in <ABA> is <as1><bs1><as1> in <ABA> ⊣

CV:

   1 : a<as>  in <as> co
   2 : b<bs>  in <bs> co
   3 : aba in <ABA> co
   4 : <as1>a<bs1>b<as1>a in <ABA> is <as1><bs1><as1> in <ABA>

SN:  aaabbbaaa in <ABA>

IR(  4 ):  aabbaa in <ABA>
IR(  4 ):  aba in <ABA>
CV:

   1 : a<as>  in <as> co
        .
        .
        .
   4 : <as1>a<bs1>b<as1>a in <ABA> is <as1><bs1><as1> in <ABA> co
   5 :  tr

SN:  aaabbaaa in <ABA>

IR(  4 ):  aabaa in <ABA>
CV:

   1 : a<as>  in <as> co
        .
        .
        .
   4 : <as1>a<bs1>b<as1>a in <ABA> is <as1><bs1><as1> in <ABA> co
   5 :  tr  co
   6 : aabaa in <ABA>
```

4.2.2.4. Example 8.
A finite automaton.
A two state, two symbol finite automaton is defined. See also 3.2.4.1.

Input :

```
⊰ a  in  <symbol> co
  b  in  <symbol> co

  <symbol><tape>  in  <tape> co

  1  in  <state> co
  2  in  <state> co

  2  in  <final state> co

  <state1><symbol1><state2> im
  <state1><symbol1><tape1>  is  <state2><tape1> co

  <state>          is ⊰ tape not accepted ⊱ co
  <final state>    is ⊰ tape accepted ⊱ co

  1 a 2 co
  1 b 1 co
  2 a 1 co
  2 b 2 ⊱ co

  1 a a  co  2 b a a
```

Output :

```
SN:
 ⊰ a in <symbol> co b in <symbol> co <symbol><tape>  in <tape> co 1 in
 <state> co 2 in <state> co 2 in <finalstate> co <state1><symbol1>
 <state2> im <state1><symbol1><tape1>  is <state2><tape1>  co <state>
 is  ⊰ tapenotaccepted ⊱  co <finalstate> is  ⊰ tapeaccepted ⊱  co
 1a2 co 1b1 co 2a1 co 2b2 ⊱
```

```
CV:

  1 : a in <symbol> co
  2 : b in <symbol> co
  3 : <symbol><tape> in <tape> co
  4 : 1 in <state> co
  5 : 2 in <state> co
  6 : 2 in <finalstate> co
  7 : <state1><symbol1><state2> im
      <state1><symbol1><tape1> is <state2><tape1> co
  8 : <state> is ⊀ tapenotaccepted ⊁ co
  9 : <finalstate> is ⊀ tapeaccepted ⊁ co
 10 : 1a2 co
 11 : 1b1 co
 12 : 2a1 co
 13 : 2b2

SN:  1aa

IR( 7 ):  2a
IR( 7 ):  1
IR( 8 ):   ⊀ tapenotaccepted ⊁
CV:

  1 : a in <symbol> co
         .
         .
         .
 13 : 2b2 co
 14 : tapenotaccepted

SN:  2baa

IR( 7 ):  2aa
IR( 7 ):  1a
IR( 7 ):  2
IR( 9 ):   ⊀ tapeaccepted ⊁
CV:

  1 : a in <symbol> co
         .
         .
         .
 13 : 2b2 co
 14 : tapenotaccepted co
 15 : tapeaccepted
```

4.2.2.5. Example 9.

A pushdown automaton for the recognition of the language $\{a^n\ b^n \mid n \geq 1\}$.
Theorem 3.2.4.2 has been applied to the pushdown automaton which is given in [22], p. 66.

Input :

```
⊀ a  in  <symbol> co
  b  in  <symbol> co

  <symbol><tape>  in  <tape> co

  p0  in  <state> co
  p1  in  <state> co
  p2  in  <state> co

  p2  in  <final state> co

  z0  in  <pd symbol> co
  z1  in  <pd symbol> co
  z2  in  <pd symbol> co

  <pd symbol><pd tape>  in  <pd tape> co

  <pd tape>                  in  <pd tapelist> co
  <pd tape>,<pd tapelist>    in  <pd tapelist> co

  <state><statelist>  in  <statelist> co

  <state1><symbol1><pd symbol1><statelist1><pd tapelist1>  im
  <state1><symbol1><tape1><pd symbol1><pd tape1>
  is
  <statelist1><tape1> <pd tapelist1>,<pd tape1>  co

  <state1><statelist1><tape1><pd tape1>,<pd tapelist1>,<pd tape2>
  is
  ( <state1><tape1><pd tape1> <pd tape2> co
    <statelist1><tape1><pd tapelist1>,<pd tape2> ) co

  <state1><tape1><pd tape1>,<pd tape2>
  is
  <state1><tape1><pd tape1> <pd tape2> co

  <statelist><final state><statelist> <pd tapelist>
  is
  ⊀ tape accepted ⊁ co
```

```
p0 a z0 p0 z2       co
p0 a z2 p0 z1 z2    co
p0 a z1 p0 z1 z1    co
p0 b z1 p1          co
p0 b z2 p2 z2       co
p1 b z1 p1          co
p1 b z2 p2 z2     } co

p0 aaabbb z0
```

Output :

SN:

```
{ a in <symbol> co b in <symbol> co <symbol><tape>  in <tape> co p0
in <state> co p1 in <state> co p2 in <state> co p2 in <finalstate>
co z0 in <pdsymbol> co z1 in <pdsymbol> co z2 in <pdsymbol> co
<pdsymbol><pdtape>  in <pdtape> co <pdtape> in <pdtapelist> co
<pdtape> ,<pdtapelist>  in <pdtapelist> co <state><statelist> in
<statelist> co <state1><symbol1><pdsymbol1><statelist1><pdtapelist1>
im <state1><symbol1><tape1> <pdsymbol1><pdtape1>  is <statelist1>
<tape1> <pdtapelist1> ,<pdtape1>  co <state1><statelist1><tape1>
<pdtape1> ,<pdtapelist1> ,<pdtape2> is  ( <state1><tape1><pdtape1>
<pdtape2>  co <statelist1><tape1><pdtapelist1> ,<pdtape2>  ) co
<state1><tape1><pdtape1> ,<pdtape2>  is <state1><tape1><pdtape1>
<pdtape2>  co <statelist> <finalstate><statelist> <pdtapelist>  is
{ tapeaccepted }  co p0az0p0z2 co p0az2p0z1z2 co p0az1p0z1z1 co
p0bz1p1 co p0bz2p2z2 co p1bz1p1 co p1bz2p2z2 }
```

CV:

```
 1 : a in <symbol> co
 2 : b in <symbol> co
 3 : <symbol><tape>  in <tape> co
 4 : p0 in <state> co
 5 : p1 in <state> co
 6 : p2 in <state> co
 7 : p2 in <finalstate> co
 8 : z0 in <pdsymbol> co
 9 : z1 in <pdsymbol> co
10 : z2 in <pdsymbol> co
11 : <pdsymbol><pdtape>  in <pdtape> co
12 : <pdtape> in <pdtapelist> co
13 : <pdtape> ,<pdtapelist>  in <pdtapelist> co
14 : <state><statelist>  in <statelist> co
```

```
15 : <state1><symbol1><pdsymbol1><statelist1><pdtapelist1>  im
     <state1><symbol1><tape1> <pdsymbol1><pdtape1>  is
     <statelist1><tape1> <pdtapelist1> ,<pdtape1>  co
16 : <state1><statelist1><tape1><pdtape1> ,<pdtapelist1> ,<pdtape2>
     is  ( <state1><tape1><pdtape1> <pdtape2>  co
     <statelist1><tape1><pdtapelist1> ,<pdtape2>  )  co
17 : <state1><tape1><pdtape1> ,<pdtape2>  is
     <state1><tape1><pdtape1> <pdtape2>  co
18 : <statelist> <finalstate><statelist> <pdtapelist>  is
     ⊀ tapeaccepted ⊁  co
19 : p0az0p0z2 co
20 : p0az2p0z1z2 co
21 : p0az1p0z1z1 co
22 : p0bz1p1 co
23 : p0bz2p2z2 co
24 : p1bz1p1 co
25 : p1bz2p2z2

SN:  p0aaabbbz0

IR( 15 ):  p0aabbbz2,
IR( 17 ):  p0aabbbz2
IR( 15 ):  p0abbbz1z2,
IR( 17 ):  p0abbbz1z2
IR( 15 ):  p0bbbz1z1,z2
IR( 17 ):  p0bbbz1z1z2
IR( 15 ):  p1bb,z1z2
IR( 17 ):  p1bbz1z2
IR( 15 ):  p1b,z2
IR( 17 ):  p1bz2
IR( 15 ):  p2z2,
IR( 18 ):   ⊀ tapeaccepted ⊁
CV:

  1 : a in <symbol> co
        .
        .
        .
 25 : p1bz2p2z2 co
 26 : tapeaccepted
```

4.2.3. Examples related to the definition of ALGOL 60.

4.2.3.1. Example 10.

Conditional expressions.
If the expression between if and then is not equal to one of the symbols true or false, it is first evaluated. If the result is true, then the value of the original expression is the value of the expression between then and else; if it is false then its value is the value of the expression after else.
An arbitrary choice has been made for the value of a simple expression(i.e. a sequence of a's and b's). If it begins with an a,it has the value true, otherwise its value is false.
Cf. chapter 5,section 22,truths 22.3 to 22.8.

Input :

```
≮ a <sexp>  in  <sexp> co
  b <sexp>  in  <sexp> co

  <sexp>  in  <exp> co

  if  <exp>  then  <exp>  else  <exp>  in  <exp> co

  if  <exp1>  then  <exp2>  else  <exp3>
  is
  if va (<exp1>)  then  <exp2>  else <exp3>  co

  if  true  then  <exp1>  else  <exp>  is  <exp1> co

  if  false  then  <exp>  else  <exp1>  is  <exp1> co

  a <sexp>  is ≮  true ≯   co
  b <sexp>  is ≮ false ≯  ≯  co

  if a then if b then a else ab else a co

  if if b then ab else ba then ab else aa
```

Output :

```
SN:
 ⊀ a<sexp>  in <sexp> co b<sexp>  in <sexp> co <sexp> in <exp> co
 if <exp> then <exp> else <exp> in <exp> co
 if <exp1> then <exp2> else <exp3> is
 if  va  ( <exp1> )  then <exp2> else <exp3> co
 if  true  then <exp1> else <exp> is <exp1> co
 if  false  then <exp> else <exp1> is <exp1> co
 a<sexp>  is  ⊀  true  ⊁  co b<sexp>  is  ⊀  false  ⊁  ⊁

CV:

   1 : a<sexp>  in <sexp> co
   2 : b<sexp>  in <sexp> co
   3 : <sexp> in <exp> co
   4 :  if <exp> then <exp> else <exp> in <exp> co
   5 :  if <exp1> then <exp2> else <exp3> is
        if  va  ( <exp1> )  then <exp2> else <exp3> co
   6 :  if  true  then <exp1> else <exp> is <exp1> co
   7 :  if  false  then <exp> else <exp1> is <exp1> co
   8 : a<sexp>  is  ⊀  true  ⊁  co
   9 : b<sexp>  is  ⊀  false  ⊁

SN:  if a then  if b then a else ab else a

TS(  5 ):  a
IR(  8 ):   ⊀  true  ⊁
IR(  5 ):   if  true  then  if b then a else ab else a
IR(  6 ):   if b then a else ab
TS(  5 ):  b
IR(  9 ):   ⊀  false  ⊁
IR(  5 ):   if  false  then a else ab
IR(  7 ):  ab
IR(  8 ):   ⊀  true  ⊁
CV:

   1 : a<sexp>  in <sexp> co
         .
         .
         .
   9 : b<sexp>  is  ⊀  false  ⊁  co
  10 :  true
```

```
SN:   if  if b then ab else ba then ab else aa

TS(  5 ):   if b then ab else ba
TS(  5 ):  b
IR(  9 ):   ⊀ false ⊁
IR(  5 ):   if  false  then ab else ba
IR(  7 ):  ba
IR(  9 ):   ⊀ false ⊁
IR(  5 ):   if  false  then ab else aa
IR(  7 ):  aa
IR(  8 ):   ⊀ true ⊁
CV:

   1 : a<sexp>  in <sexp> co
         .
         .
         .
   9 : b<sexp>  is  ⊀ false ⊁  co
  10 :  true  co
  11 :  true
```

4.2.3.2. Example 11.

Definition of the logical operators ¬ , ∧ , ∨. Operations upon true and false by the operators ¬ , ∧ , ∨ , along with their priority rules and the meaning of parentheses are defined. This example demonstrates the principle for the definition of boolean expressions. Cf. chapter 5, section 22.

Input :

```
⊀ true  in  <bprimary> co
  false in  <bprimary> co

  (<bexp>)  in  <bprimary> co

  <bprimary>    in  <bsecondary> co
  ¬ <bprimary>  in  <bsecondary> co

  <bsecondary>                in  <bfactor> co
  <bfactor> ∧ <bsecondary>    in  <bfactor> co

  <bfactor>                   in  <bexp> co
  <bexp> ∨ <bfactor>          in  <bexp> co

  (<bexp1>)  is  <bexp1> co

  ¬ <bprimary1>  is  ¬ va ( <bprimary1> ) co

  <bfactor1> ∧ <bsecondary1>
  is
  va (<bfactor1>) ∧ va ( <bsecondary1>) co

  <bexp1> ∨ <bfactor1>
  is
  va (<bexp1>) ∨ va (<bfactor1>) co

  ¬ true   is  false co
  ¬ false  is  true  co

  true   ∧  true   is  true   co
  true   ∧  false  is  false  co
  false  ∧  true   is  false  co
  false  ∧  false  is  false  co

  true   ∨  true   is  true   co
  true   ∨  false  is  true   co
  false  ∨  true   is  true   co
  false  ∨  false  is  false  ⊁  co

  ¬ true ∨ false  ∧ false  co  ( true ∨ false )  ∧ true
```

Output :

SN:
⊀ true in <bprimary> co false in <bprimary> co (<bexp>) in <bprimary> co <bprimary> in <bsecondary> co ¬<bprimary> in <bsecondary> co <bsecondary> in <bfactor> co <bfactor>∧<bsecondary> in <bfactor> co <bfactor> in <bexp> co <bexp>∨<bfactor> in <bexp> co (<bexp1>) is <bexp1> co ¬<bprimary1> is ¬ va (<bprimary1>) co <bfactor1>∧<bsecondary1> is va (<bfactor1>) ∧ va (<bsecondary1>) co <bexp1>∨<bfactor1> is va (<bexp1>) ∨ va (<bfactor1>) co ¬ true is false co ¬ false is true co true ∧ true is true co true ∧ false is false co false ∧ true is false co false ∧ false is false co true ∨ true is true co true ∨ false is true co false ∨ true is true co false ∨ false is false ⊁

CV:

1 : true in <bprimary> co
2 : false in <bprimary> co
3 : (<bexp>) in <bprimary> co
4 : <bprimary> in <bsecondary> co
5 : ¬<bprimary> in <bsecondary> co
6 : <bsecondary> in <bfactor> co
7 : <bfactor>∧<bsecondary> in <bfactor> co
8 : <bfactor> in <bexp> co
9 : <bexp>∨<bfactor> in <bexp> co
10 : (<bexp1>) is <bexp1> co
11 : ¬<bprimary1> is ¬ va (<bprimary1>) co
12 : <bfactor1>∧<bsecondary1> is
va (<bfactor1>) ∧ va (<bsecondary1>) co
13 : <bexp1>∨<bfactor1> is
va (<bexp1>) ∨ va (<bfactor1>) co
14 : ¬ true is false co
15 : ¬ false is true co
16 : true ∧ true is true co
17 : true ∧ false is false co
18 : false ∧ true is false co
19 : false ∧ false is false co
20 : true ∨ true is true co
21 : true ∨ false is true co
22 : false ∨ true is true co
23 : false ∨ false is false

```
SN:  ¬ true ∨ false ∧ false

TS( 13 ):  ¬ true
IR( 14 ):   false
TS( 13 ):   false ∧ false
IR( 19 ):   false
IR( 13 ):   false ∨ false
IR( 23 ):   false
CV:

  1 :  true  in <bprimary> co
        .
        .
        .
 23 :  false ∨ false  is  false  co
 24 :  false

SN:  ( true ∨ false )∧ true

TS( 12 ):  ( true ∨ false )
IR( 10 ):   true ∨ false
IR( 21 ):   true
TS( 12 ):   true
IR( 12 ):   true ∧ true
IR( 16 ):   true
CV:

  1 :  true  in <bprimary> co
        .
        .
        .
 23 :  false ∨ false  is  false  co
 24 :  false  co
 25 :  true
```

4.2.3.3.Example 12.

Integer addition and subtraction and assignment statements. The principle for the treatment of assignment statements is given.The addition and subtraction of integers are defined.Cf. [41] ,p.18 and chapter 5, section 22.

We have chosen 4 as the base for the number system in order to reduce the time needed for the execution of this example.

Input :

```
⊀ 0  in  <di> co
  1  in  <di> co
  2  in  <di> co
  3  in  <di> co

  <di><ui>  in  <ui> co
  <pm><ui>  in  <in> co

  0<ze>  in  <ze> co

  +  in  <pm> co
  -  in  <pm> co

 x <id>  in  <id> co

 <id>  in  <primary> co
 <ui>  in  <primary> co

 <pm><primary>          in  <exp> co
 <exp><pm><primary>     in  <exp> co

 <pm1><primary1>  is  <pm1> va (<primary1>) co

 <exp1><pm1><primary1>  is  va (<exp1>) <pm1> va (<primary1>) co

 <id1>:= <exp1>  is  <id1> := va (<exp1>) co

 <id1>:= <in1>   is ⊀ <id1> is <in1> ⊁ co

  -<ui1> + <ui2>    is  <ui2> - <ui1> co
  <in1> - - <ui1>   is  <in1> + <ui1> co
  <in1> + - <ui1>   is  <in1> - <ui1> co
  - <ui1> - <ui2>   is  - va ( <ui1> + <ui2> ) co

  <ui1><di1><pm1><ui2><di2>
  is
  va (<ui1><pm1><ui2> ) 0 +  va (<di1><pm1><di2>) co

  <ui1><di1><pm1><di2>  is  <ui1> 0 + va (<di1><pm1><di2>) co
  <di1><pm1><ui1><di2>  is  <pm1><ui1> 0  + va (<di1><pm1><di2>) co
```

```
<ui1> 0 + <di1>  is  <ui1><di1> co
<di1> + <ui1> 0  is  <ui1><di1> co

<ui1> 0 - <di1>  is  va (<ui1> - 1) 0  + va (10 - <di1> ) co
10 - <di1>  is  3 - va ( <di1> - 1) co

<di1><pm1><di2>  is  va (<di1><pm1> 1) <pm1> va (<di2> - 1) co

<ui1><pm><ze>  is  <ui1> co
<ze><pm1><ui1> is  <pm1><ui1> co

0 + 1 is 1  co
1 + 1 is 2  co
2 + 1 is 3  co
3 + 1 is 10 co

1 - 1 is 0 co
2 - 1 is 1 co
3 - 1 is 2 co

+<ui1>  is { <ui1>  } co
-<ui1>  is { -<ui1> }  } co

x:= 1 + 2 + 3 co xx := x - 10 co xx := xx + x
```

Output :

```
SN:
{ 0 in <di> co 1 in <di> co 2 in <di> co 3 in <di> co <di><ui>  in
<ui> co <pm> <ui> in <in> co 0<ze>  in <ze> co + in <pm> co - in
<pm> co x<id>  in <id> co <id> in <primary> co <ui> in <primary> co
<pm> <primary> in <exp> co <exp><pm><primary> in <exp> co <pm1>
<primary1> is <pm1> va ( <primary1> )  co <exp1><pm1><primary1> is
va  ( <exp1> ) <pm1> va  ( <primary1> )  co <id1>:=<exp1> is <id1>:=
va  ( <exp1> )  co <id1>:=<in1> is  { <id1> is <in1> }  co -<ui1>+
<ui2> is <ui2>-<ui1> co <in1>--<ui1> is <in1>+<ui1> co <in1>+-<ui1>
is <in1>-<ui1> co -<ui1>-<ui2> is - va  ( <ui1>+<ui2> )  co <ui1>
<di1><pm1><ui2><di2> is  va  ( <ui1><pm1><ui2> ) 0+ va  ( <di1><pm1>
<di2> )  co <ui1><di1><pm1><di2> is <ui1>0+ va  ( <di1><pm1><di2>
)  co <di1><pm1><ui1><di2> is <pm1><ui1>0+ va  ( <di1><pm1><di2> )
co <ui1>0+<di1> is <ui1><di1> co <di1>+<ui1>0 is <ui1><di1> co <ui1>
0-<di1> is  va  ( <ui1>-1 ) 0+ va  ( 10-<di1> )  co 10-<di1> is 3
- va  ( <di1>-1 )  co <di1><pm1><di2> is  va  ( <di1><pm1>1 ) <pm1>
va  ( <di2>-1 )  co <ui1><pm><ze> is <ui1> co <ze><pm1><ui1> is
<pm1><ui1> co 0+1 is 1 co 1+1 is 2 co 2+1 is 3 co 3+1 is 10 co 1-1
is 0 co 2-1 is 1 co 3-1 is 2 co +<ui1> is  { <ui1> }  co -<ui1> is
{ -<ui1> }  }
```

CV:

```
 1 : 0 in <di> co
 2 : 1 in <di> co
 3 : 2 in <di> co
 4 : 3 in <di> co
 5 : <di><ui>  in <ui> co
 6 : <pm> <ui> in <in> co
 7 : 0<ze>  in <ze> co
 8 : + in <pm> co
 9 : - in <pm> co
10 : x<id>  in <id> co
11 : <id> in <primary> co
12 : <ui> in <primary> co
13 : <pm> <primary> in <exp> co
14 : <exp><pm><primary> in <exp> co
15 : <pm1><primary1> is <pm1> va ( <primary1> ) co
16 : <exp1><pm1><primary1> is
     va ( <exp1> ) <pm1> va ( <primary1> ) co
17 : <id1>:=<exp1> is <id1>:= va ( <exp1> ) co
18 : <id1>:=<in1> is { <id1> is <in1> } co
19 : -<ui1>+<ui2> is <ui2>-<ui1> co
20 : <in1>--<ui1> is <in1>+<ui1> co
21 : <in1>+-<ui1> is <in1>-<ui1> co
22 : -<ui1>-<ui2> is - va ( <ui1>+<ui2> ) co
23 : <ui1><di1><pm1><ui2><di2> is
     va ( <ui1><pm1><ui2> ) 0+ va ( <di1><pm1><di2> ) co
24 : <ui1><di1><pm1><di2> is <ui1>0+ va ( <di1><pm1><di2> ) co
25 : <di1><pm1><ui1><di2> is <pm1><ui1>0+ va ( <di1><pm1><di2> ) co
26 : <ui1>0+<di1> is <ui1><di1> co
27 : <di1>+<ui1>0 is <ui1><di1> co
28 : <ui1>0-<di1> is va ( <ui1>-1 ) 0+ va ( 10-<di1> ) co
29 : 10-<di1> is 3- va ( <di1>-1 ) co
30 : <di1><pm1><di2> is va ( <di1><pm1>1 ) <pm1> va ( <di2>-1 ) co
31 : <ui1><pm><ze> is <ui1> co
32 : <ze><pm1><ui1> is <pm1><ui1> co
33 : 0+1 is 1 co
34 : 1+1 is 2 co
35 : 2+1 is 3 co
36 : 3+1 is 10 co
37 : 1-1 is 0 co
38 : 2-1 is 1 co
39 : 3-1 is 2 co
40 : +<ui1> is { <ui1> } co
41 : -<ui1> is { -<ui1> }
```

```
SN:  x:=1+2+3

TS( 17 ):  1+2+3
TS( 16 ):  1+2
TS( 30 ):  1+1
IR( 34 ):  2
TS( 30 ):  2-1
IR( 38 ):  1
IR( 30 ):  2+1
IR( 35 ):  3
TS( 16 ):  3
IR( 16 ):  3+3
TS( 30 ):  3+1
IR( 36 ):  10
TS( 30 ):  3-1
IR( 39 ):  2
IR( 30 ):  10+2
IR( 26 ):  12
IR( 17 ):  x:=12
IR( 18 ):   ⊀ x is 12 ⊁
CV:

  1 : 0 in <di> co
        .
        .
        .
 41 : -<ui1> is  ⊀ -<ui1> ⊁  co
 42 : x is 12

SN:  xx:=x-10

TS( 17 ):  x-10
TS( 16 ):  x
IR( 42 ):  12
TS( 16 ):  10
IR( 16 ):  12-10
TS( 23 ):  1-1
IR( 37 ):  0
TS( 23 ):  2-0
IR( 31 ):  2
IR( 23 ):  00+2
IR( 32 ):  +2
IR( 40 ):   ⊀ 2 ⊁
IR( 17 ):  xx:=2
IR( 18 ):   ⊀ xx is 2 ⊁
```

```
CV:

  1 : 0 in <di> co
        .
        .
        .
 41 : -<ui1> is  ≮ -<ui1> ≯  co
 42 : x is 12 co
 43 : xx is 2

SN:  xx:=xx+x

TS( 17 ):  xx+x
TS( 16 ):  xx
IR( 43 ):  2
TS( 16 ):  x
IR( 42 ):  12
IR( 16 ):  2+12
TS( 25 ):  2+2
TS( 30 ):  2+1
IR( 35 ):  3
TS( 30 ):  2-1
IR( 38 ):  1
IR( 30 ):  3+1
IR( 36 ):  10
IR( 25 ):  +10+10
TS( 16 ):  +10
IR( 40 ):   ≮ 10 ≯
TS( 16 ):  10
IR( 16 ):  10+10
TS( 23 ):  1+1
IR( 34 ):  2
TS( 23 ):  0+0
IR( 32 ):  +0
IR( 40 ):   ≮ 0 ≯
IR( 23 ):  20+0
IR( 31 ):  20
IR( 17 ):  xx:=20
IR( 18 ):   ≮ xx is 20 ≯
CV:

  1 : 0 in <di> co
        .
        .
        .
 41 : -<ui1> is  ≮ -<ui1> ≯  co
 42 : x is 12 co
 43 : xx is 2 co
 44 : xx is 20
```

4.2.3.4. Example 13.

Goto statements.

This example demonstrates the principle of the definition of goto statements. In a "prescan" each statement is numbered and supplied with the number of its successor. After the prescan is finished the actual evaluation of the "program" is started by the evaluation of the first number. The evaluation of a goto statement referring to a certain label leads to the evaluation of the number of the statement which is labelled by this label. Many more details of the prescan mechanism for ALGOL 60 (which is in fact much more complicated, mainly because of the block structure) are given in chapter 6.

Input :

```
⊰ S  in  <statement> co
  T  in  <statement> co
  U  in  <statement> co

  goto <label>  in  <statement> co

  <label> : <statement>  in  <statement> co

  <statement>                        in  <statement list> co
  <statement> ; <statement list>     in  <statement list> co

  1  in  <label> co
  2  in  <label> co
  3  in  <label> co

  a<as>  in  <as> co

  begin  <statement list1>  end
  is
  ( a : <statement list1>  co a ) co

  <as1> : <statement1> ; <statement list1>
  is
  (⊰ <as1> is ( <statement1> co <as1>a ) ⊱ co
    <as1>a : <statement list1> ) co

  <as1> : goto <label1>;<statement list1>
  is
  (⊰ <as1> is goto <label1> ⊱ co <as1>a : <statement list1> ) co

  <as1> : <statement1> is ⊰ <as1> is <statement1> ⊱ co
```

```
<as1> : <label1> : <statement list1>
is
( ⊀ <label1> is <as1> ⊁ co <as1> : <statement list1> ) co

goto <label1> is <label1>  ⊁ co

begin S ; T ; goto 1 ; S ; 1 : U end
```

Output :

```
SN:
 ⊀ S in <statement> co T in <statement> co U in <statement> co  goto
 <label> in <statement> co <label>:<statement> in <statement> co
 <statement> in <statementlist> co <statement>;<statementlist> in
 <statementlist> co 1 in <label> co 2 in <label> co 3 in <label> co
 a<as>  in <as> co  begin <statementlist1> end  is  ( a:
 <statementlist1> co a )  co <as1>:<statement1>;<statementlist1> is
 (  ⊀ <as1> is  ( <statement1> co <as1>a )  ⊁  co <as1>a:
 <statementlist1> )  co <as1>: goto <label1>;<statementlist1> is  (
 ⊀ <as1> is  goto <label1> ⊁  co <as1>a:<statementlist1> )  co <as1>
 :<statement1> is  ⊀ <as1> is <statement1> ⊁  co <as1>:<label1>:
 <statementlist1> is  (  ⊀ <label1> is <as1> ⊁ co <as1>:
 <statementlist1> )  co  goto <label1> is <label1> ⊁

CV:

  1 : S in <statement> co
  2 : T in <statement> co
  3 : U in <statement> co
  4 :  goto <label> in <statement> co
  5 : <label>:<statement> in <statement> co
  6 : <statement> in <statementlist> co
  7 : <statement>;<statementlist> in <statementlist> co
  8 : 1 in <label> co
  9 : 2 in <label> co
 10 : 3 in <label> co
 11 : a<as>  in <as> co
 12 :  begin <statementlist1> end  is
     ( a:<statementlist1> co a )  co
 13 : <as1>:<statement1>;<statementlist1> is
     (  ⊀ <as1> is  ( <statement1> co <as1>a )  ⊁  co
     <as1>a:<statementlist1> )  co
 14 : <as1>: goto <label1>;<statementlist1> is
     (  ⊀ <as1> is  goto <label1> ⊁  co
     <as1>a:<statementlist1> )  co
 15 : <as1>:<statement1> is  ⊀ <as1> is <statement1> ⊁  co
 16 : <as1>:<label1>:<statementlist1> is
     (  ⊀ <label1> is <as1> ⊁  co <as1>:<statementlist1> )  co
 17 :  goto <label1> is <label1>
```

```
SN:   begin S;T; goto 1;S;1:U end

IR( 12 ):  a:S;T; goto 1;S;1:U
IR( 13 ):   ⊀ a is ( S co aa ) ⊁
CV:

  1 : S in <statement> co
        .
        .
        .
 17 :  goto <label1> is <label1> co
 18 : a is  ( S co aa )

IR( 13 ):  aa:T; goto 1,S;1:U
IR( 13 ):   ⊀ aa is ( T co aaa ) ⊁
CV:

  1 : S in <statement> co
        .
        .
        .
 17 :  goto <label1> is <label1> co
 18 : a is  ( S co aa )  co
 19 : aa is  ( T co aaa )

IR( 13 ):  aaa: goto 1;S;1:U
IR( 14 ):   ⊀ aaa is  goto 1 ⊁
CV:

  1 : S in <statement> co
        .
        .
        .
 17 :  goto <label1> is <label1> co
 18 : a is  ( S co aa )  co
 19 : aa is  ( T co aaa )  co
 20 : aaa is  goto 1

IR( 14 ):  aaaa:S;1:U
IR( 13 ):   ⊀ aaaa is  ( S co aaaaa ) ⊁
CV:

  1 : S in <statement> co
        .
        .
        .
 17 :  goto <label1> is <label1> co
 18 : a is  ( S co aa )  co
 19 : aa is  ( T co aaa )  co
 20 : aaa is  goto 1 co
 21 : aaaa is  ( S co aaaaa )
```

```
IR( 13 ):  aaaaa:1:U
IR( 16 ):   ⊀ 1 is aaaaa ⊁
CV:

   1 : S in <statement> co
         .
         .
         .
  17 :  goto <label1> is <label1> co
  18 : a is  ( S co aa )  co
  19 : aa is  ( T co aaa )  co
  20 : aaa is  goto 1 co
  21 : aaaa is  ( S co aaaaa )  co
  22 : 1 is aaaaa

IR( 16 ):  aaaaa:U
IR( 15 ):   ⊀ aaaaa is U ⊁
CV:

   1 : S in <statement> co
         .
         .
         .
  17 :  goto <label1> is <label1> co
  18 : a is  ( S co aa )  co
  19 : aa is  ( T co aaa )  co
  20 : aaa is  goto 1 co
  21 : aaaa is  ( S co aaaaa )  co
  22 : 1 is aaaaa co
  23 : aaaaa is U

IR( 12 ):  a
IR( 18 ):  S
CV:

   1 : S in <statement> co
         .
         .
         .
  17 :  goto <label1> is <label1> co
  18 : a is  ( S co aa )  co
  19 : aa is  ( T co aaa )  co
  20 : aaa is  goto 1 co
  21 : aaaa is  ( S co aaaaa )  co
  22 : 1 is aaaaa co
  23 : aaaaa is U co
  24 : S
```

```
IR( 18 ):  aa
IR( 19 ):  T
CV:

  1 : S in <statement> co
        .
        .
        .
 17 :  goto <label1> is <label1> co
 18 : a is  ( S co aa )  co
 19 : aa is  ( T co aaa )  co
 20 : aaa is  goto 1 co
 21 : aaaa is  ( S co aaaaa )  co
 22 : 1 is aaaaa co
 23 : aaaaa is U co
 24 : S co
 25 : T

IR( 19 ):  aaa
IR( 20 ):  goto 1
IR( 17 ):  1
IR( 22 ):  aaaaa
IR( 23 ):  U
CV:

  1 : S in <statement> co
        .
        .
        .
 17 :  goto <label1> is <label1> co
 18 : a is  ( S co aa )  co
 19 : aa is  ( T co aaa )  co
 20 : aaa is  goto 1 co
 21 : aaaa is  ( S co aaaaa )  co
 22 : 1 is aaaaa co
 23 : aaaaa is U co
 24 : S co
 25 : T co
 26 : U
```

4.2.4. Wang's algorithm for the propositional calculus.

Example 14. This example defines the well known algorithm of Wang for the propositional calculus [45,36]. Truth 15 is Wang's rule P1, truths 16,17,...,25 correspond to his rules P2a,P2b,...,P6b.The equal sign replaces Wang's arrow.
A form(ula) is valid if and only if the evaluation of the simple name that denotes the formula does not lead to the addition to V of a truth different from "valid".
The idea of using the metalanguage for the definition of this algorithm was taken from PANON 1B, see [9].

Input :

```
≮ P  in  <atomic form> co
  Q  in  <atomic form> co
  R  in  <atomic form> co
  S  in  <atomic form> co
  T  in  <atomic form> co

  <atomic form><at form seq>  in  <at form seq> co

  <atomic form>             in  <form> co
  (¬ <form>)                in  <form> co
  (<form> ∧ <form>)         in  <form> co
  (<form> ∨ <form>)         in  <form> co
  (<form> ⊃ <form>)         in  <form> co
  (<form> ≡ <form>)         in  <form> co

  <form><form seq>  in  <form seq> co

  <at form seq> = <at form seq>  is  ≮ non valid ≯ co

  <at form seq><atomic form1><at form seq>
  =
  <at form seq><atomic form1><at form seq>
  is
  ≮ valid ≯ co
```

<at form seq1> = <at form seq2>(¬ <form1>)<form seq1>
is
<form1><at form seq1> = <at form seq2> <form seq1> co

<at form seq1>(¬ <form1>)<form seq1> = <form seq2>
is
<at form seq1> <form seq1> = <form seq2><form1> co

<at form seq1> = <at form seq2>(<form1> ∧ <form2>)<form seq1>
is
[<at form seq1> = <at form seq2><form1><form seq1> co
<at form seq1> = <at form seq2><form2><form seq1>] co

<at form seq1>(<form1> ∧ <form2>)<form seq1> = <form seq2>
is
<at form seq1><form1><form2><form seq1> = <form seq2> co

<at form seq1> = <at form seq2>(<form1> ∨ <form2>)<form seq1>
is
<at form seq1> = <at form seq2><form1><form2><form seq1> co

<at form seq1>(<form1> ∨ <form2>)<form seq1> = <form seq2>
is
[<at form seq1><form1><form seq1> = <form seq2> co
<at form seq1><form2><form seq1> = <form seq2>] co

<at form seq1> = <at form seq2>(<form1> ⌉ <form2>)<form seq1>
is
<at form seq1><form1> = <at form seq2><form2><form seq1> co

<at form seq1>(<form1> ⌉ <form2>)<form seq1> = <form seq2>
is
[<at form seq1><form2><form seq1> = <form seq2> co
<at form seq1> <form seq1> = <form seq2><form1>] co

<at form seq1> = <at form seq2>(<form1> = <form2>)<form seq1>
is
[<form1><at form seq1> = <at form seq2><form2><form seq1> co
<form2><at form seq1> = <at form seq2><form1><form seq1>] co

<at form seq1>(<form1> = <form2>)<form seq1> = <form seq2>
is
[<form1><form2><at form seq1> <form seq1> = <form seq2> co
<at form seq1> <form seq1> = <form seq2><form1><form2>] ⊁ co

= (((¬ P) ∧ (¬ Q)) ⌉ (P = Q)) co

= ((P ∨ Q) ⌉ (P ∧ Q))

Output :

SN:

‡ P in <atomicform> co Q in <atomicform> co R in <atomicform> co S in <atomicform> co T in <atomicform> co <atomicform><atformseq> in <atformseq> co <atomicform> in <form> co (¬<form>) in <form> co (<form>∧<form>) in <form> co (<form>∨<form>) in <form> co (<form> ⌉ <form>) in <form> co (<form> = <form>) in <form> co <form><formseq> in <formseq> co <atformseq> =<atformseq> is ‡ nonvalid ‡ co <atformseq> <atomicform1><atformseq> =<atformseq> <atomicform1> <atformseq> is ‡ valid ‡ co <atformseq1> =<atformseq2> (¬<form1>) <formseq1> is <form1><atformseq1> =<atformseq2> <formseq1> co <atformseq1> (¬<form1>)<formseq1> =<formseq2> is <atformseq1> <formseq1> =<formseq2> <form1> co <atformseq1> =<atformseq2> (<form1> ∧<form2>)<formseq1> is (<atformseq1> =<atformseq2> <form1><formseq1> co <atformseq1> =<atformseq2> <form2><formseq1>) co <atformseq1> (<form1>∧<form2>)<formseq1> =<formseq2> is <atformseq1> <form1> <form2><formseq1> =<formseq2> co <atformseq1> =<atformseq2> (<form1> ∨<form2>)<formseq1> is <atformseq1> =<atformseq2> <form1><form2> <formseq1> co <atformseq1> (<form1>∨<form2>)<formseq1> =<formseq2> is (<atformseq1> <form1><formseq1> =<formseq2> co <atformseq1> <form2><formseq1> =<formseq2>) co <atformseq1> =<atformseq2> (<form1> ⌉ <form2>)<formseq1> is <atformseq1> <form1>=<atformseq2> <form2><formseq1> co <atformseq1> (<form1> ⌉ <form2>)<formseq1> = <formseq2> is (<atformseq1> <form2><formseq1> =<formseq2> co <atformseq1> <formseq1> =<formseq2> <form1>) co <atformseq1> = <atformseq2> (<form1> = <form2>)<formseq1> is (<form1><atformseq1> =<atformseq2> <form2><formseq1> co <form2><atformseq1> =<atformseq2> <form1><formseq1>) co <atformseq1> (<form1> = <form2>)<formseq1> =<formseq2> is (<form1><form2><atformseq1> <formseq1> =<formseq2> co <atformseq1> <formseq1> =<formseq2> <form1><form2>) ‡

CV:

1 : P in <atomicform> co
2 : Q in <atomicform> co
3 : R in <atomicform> co
4 : S in <atomicform> co
5 : T in <atomicform> co
6 : <atomicform><atformseq> in <atformseq> co
7 : <atomicform> in <form> co
8 : (¬<form>) in <form> co
9 : (<form>∧<form>) in <form> co
10 : (<form>∨<form>) in <form> co
11 : (<form> ⌉ <form>) in <form> co
12 : (<form> = <form>) in <form> co
13 : <form><formseq> in <formseq> co

```
14 : <atformseq> =<atformseq> is ⊀ nonvalid ⊁ co
15 : <atformseq> <atomicform1><atformseq> =
     <atformseq> <atomicform1><atformseq> is ⊀ valid ⊁ co
16 : <atformseq1> =<atformseq2> (¬<form1>)<formseq1> is
     <form1><atformseq1> =<atformseq2> <formseq1> co
17 : <atformseq1> (¬<form1>)<formseq1> =<formseq2> is
     <atformseq1> <formseq1> =<formseq2> <form1> co
18 : <atformseq1> =<atformseq2> (<form1>∧<form2>)<formseq1> is
     ( <atformseq1> =<atformseq2> <form1><formseq1> co
     <atformseq1> =<atformseq2> <form2><formseq1> ) co
19 : <atformseq1> (<form1>∧<form2>)<formseq1> =<formseq2> is
     <atformseq1> <form1><form2><formseq1> =<formseq2> co
20 : <atformseq1> =<atformseq2> (<form1>∨<form2>)<formseq1> is
     <atformseq1> =<atformseq2> <form1><form2><formseq1> co
21 : <atformseq1> (<form1>∨<form2>)<formseq1> =<formseq2> is
     ( <atformseq1> <form1><formseq1> =<formseq2> co
     <atformseq1> <form2><formseq1> =<formseq2> ) co
22 : <atformseq1> =<atformseq2> (<form1> ⊃ <form2>)<formseq1> is
     <atformseq1> <form1>=<atformseq2> <form2><formseq1> co
23 : <atformseq1> (<form1> ⊃ <form2>)<formseq1> =<formseq2> is
     ( <atformseq1> <form2><formseq1> =<formseq2> co
     <atformseq1> <formseq1> =<formseq2> <form1> ) co
24 : <atformseq1> =<atformseq2> (<form1> = <form2>)<formseq1> is
     ( <form1><atformseq1> =<atformseq2> <form2><formseq1> co
     <form2><atformseq1> =<atformseq2> <form1><formseq1> ) co
25 : <atformseq1> (<form1> = <form2>)<formseq1> =<formseq2> is
     ( <form1><form2><atformseq1> <formseq1> =<formseq2> co
     <atformseq1> <formseq1> =<formseq2> <form1><form2> )

SN:  =(((¬P)∧(¬Q)) ⊃ (P = Q))

IR( 22 ):  ((¬P)∧(¬Q))=(P = Q)
IR( 19 ):  (¬P)(¬Q)=(P = Q)
IR( 17 ):  (¬Q)=(P = Q)P
IR( 17 ):  =(P = Q)PQ
IR( 24 ):  P=QPQ
IR( 15 ):   ⊀ valid ⊁
CV:

 1 : P in <atomicform> co
       .
       .
       .
25 : <atformseq1> (<form1> = <form2>)<formseq1> =<formseq2> is
     ( <form1><form2><atformseq1> <formseq1> =<formseq2> co
     <atformseq1> <formseq1> =<formseq2> <form1><form2> ) co
26 : valid
```

```
IR( 24 ):  Q=PPQ
IR( 15 ):   ⊀ valid ⊁
CV:

  1 : P in <atomicform> co
       .
       .
       .
 25 : <atformseq1> (<form1> = <form2>)<formseq1> =<formseq2>  is
      ( <form1><form2><atformseq1> <formseq1> =<formseq2>  co
      <atformseq1> <formseq1> =<formseq2> <form1><form2> )  co
 26 : valid co
 27 : valid

SN:  =((P∨Q) ¬ (P∧Q))

IR( 22 ):  (P∨Q)=(P∧Q)
IR( 21 ):  P=(P∧Q)
IR( 18 ):  P=P
IR( 15 ):   ⊀ valid ⊁
CV:

  1 : P in <atomicform> co
       .
       .
       .
 25 : <atformseq1> (<form1> = <form2>)<formseq1> =<formseq2>  is
      ( <form1><form2><atformseq1> <formseq1> =<formseq2>  co
      <atformseq1> <formseq1> =<formseq2> <form1><form2> )  co
 26 : valid co
 27 : valid co
 28 : valid

IR( 18 ):  P=Q
IR( 14 ):   ⊀ nonvalid ⊁
CV:

  1 : P in <atomicform> co
       .
       .
       .
 25 : <atformseq1> (<form1> = <form2>)<formseq1> =<formseq2>  is
      ( <form1><form2><atformseq1> <formseq1> =<formseq2>  co
      <atformseq1> <formseq1> =<formseq2> <form1><form2> )  co
 26 : valid co
 27 : valid co
 28 : valid co
 29 : nonvalid
```

```
IR( 21 ):  Q=(P∧Q)
IR( 18 ):  Q=P
IR( 14 ):   ⊀ nonvalid ⊁
CV:

  1 : P in <atomicform> co
        .
        .
        .
 25 : <atformseq1> (<form1> = <form2>)<formseq1> =<formseq2>  is
      ( <form1><form2><atformseq1> <formseq1> =<formseq2>  co
      <atformseq1> <formseq1> =<formseq2> <form1><form2> )  co
 26 : valid co
 27 : valid co
 28 : valid co
 29 : nonvalid co
 30 : nonvalid

IR( 18 ):  Q=Q
IR( 15 ):   ⊀ valid ⊁
CV:

  1 : P in <atomicform> co
        .
        .
        .
 25 : <atformseq1> (<form1> = <form2>)<formseq1> =<formseq2>  is
      ( <form1><form2><atformseq1> <formseq1> =<formseq2>  co
      <atformseq1> <formseq1> =<formseq2> <form1><form2> )  co
 26 : valid co
 27 : valid co
 28 : valid co
 29 : nonvalid co
 30 : nonvalid co
 31 : valid
```

CHAPTER 5

DEFINITION OF ALGOL 60

In this chapter we give the definition of ALGOL 60 by means of a metaprogram.
An explanation of this definition follows in chapter 6.

For typographical reasons, the ALGOL 60 symbols ÷ and ⊃ are denoted here by $\underline{:}$ and $\underline{\urcorner}$.

The numbers to the left of the truths and the headings of the sections are not to be interpreted as part of the metaprogram; they are introduced only for easier reference in chapter 6.

" Undefined values " .

0.1	<sequence of basic and aux term symbols>			is o	co
0.2	<ass st>	in	<decl ass st>	is o	co
0.3	<dexp>	in	<decl dexp>	is o	co
0.4	<proc st>	in	<decl proc st>	is o	co
0.5	<block>	in	<decl block>	is o	co
0.6	<bplist>	in	<decl bplist>	is o	co
0.7	<switch list>	in	<decl switch list>	is o	co

Syntax of a program.

1.1	<ass st>	in	<unlabelled basic st>	co
1.2	goto <dexp>	in	<unlabelled basic st>	co
1.3	<proc st>	in	<unlabelled basic st>	co
1.4	<unlabelled basic st>	in	<basic st>	co
1.5	<label> : <basic st>	in	<basic st>	co
1.6	<basic st>	in	<unc st>	co
1.7	<compound st>	in	<unc st>	co
1.8	<block>	in	<unc st>	co
1.9	if <bexp> then <unc st>	in	<cond st>	co
1.10	if <bexp> then <for st>	in	<cond st>	co
1.11	if <bexp> then <unc st> else <st>	in	<cond st>	co
1.12	<label> : <cond st>	in	<cond st>	co
1.13	<unc st>	in	<st>	co
1.14	<cond st>	in	<st>	co
1.15	<for st>	in	<st>	co
1.16	<st>	in	<st list>	co
1.17	<st> ; <st list>	in	<st list>	co
1.18	begin <st list> end	in	<compound st>	co
1.19	<label> : <compound st>	in	<compound st>	co

1.20 <type declaration> in <declaration> co
1.21 <array declaration> in <declaration> co
1.22 <switch declaration> in <declaration> co
1.23 <procedure declaration> in <declaration> co

1.24 <declaration> ; ≤decl list≥ in <decl list> co

1.25 begin <decl list>≤st list≥ end in <block> co
1.26 <label> : <block> in <block> co

1.27 ≤decl list≥ ≤st list≥ end in <block tail> co

1.28 <int var>:= <for list> in <ext st list> co
1.29 <st list> in <ext st list> co
1.30 ≤ext st list≥;≤ext st list≥ in <ext st list> co

1.31 ; ≤ext st list≥ in <special st list> co

1.32 ≤special st list≥ end in <block end> co

1.33 <compound st> in <program> co
1.34 <block> in <program> co

Value of a program.

```
2.1     <program1>
        is
        ( ≮ b c ≯ co ≮ f g ≯ co
           b c a : integer dummy 1; boolean dummy 2;
                   integer procedure sign (f); value f; integer f;
                   sign := if f > 0 then 1 else if f = 0 then 0
                                                         else - 1;
                   <program1> end  co
          b c a  )  co
```

2.2 <sequence of basic and aux term symbols><aux id>
<sequence of basic and aux term symbols>
is o co

2.3 <sequence of basic and aux term symbols><aux label>
<sequence of basic and aux term symbols>
is o co

Syntax of block number and program point.

3.1 a <as> in <as> co

3.2 b <bs> in <bs> co

3.3 <bs> c in <bc> co

3.4 <bc><bcs> in <bcs> co

3.5 d <bcs><dbcs> in <dbcs> co

3.6 <bcs><dbcs> in <bn> co

3.7 <bcs><as> in <p> co

Prescan declarations.

4.1 <p1> : <declaration1> ; <block tail1>
is
(<declaration1><p1> 1 co
 ⊀ <p1> is (<declaration1><p1> 2 co
 ⊀ <p1> is (
 ⊀ t <p1> is (<declaration1><p1> 4 co
 t <p1> a) ⊁ co
 <p1> a) ⊁ co
 <p1> a) ⊁ co
 <p1> a : <block tail1>) co

4.2 <p1> : <own1><type1><id1>,<id list1> ; <block tail1>
is
<p1> : <own1><type1><id1>;
<own1><type1><idlist1> ; <block tail1> co

4.3 <p1> : <own1><type1> array <array segment1>,
<array list1> ; <block tail1>
is
<p1> : <own1><type1> array <array segment1>;
<own1><type1> array <array list1> ; <block tail1> co

Prescan statements.

```
5.1     <p1> : ; <st list1> end
        is
        <p1> :   <st list1> end  co

5.2     <p1> : begin  <st list1> end <block end1>
        is
        <p1> : <st list1><block end1>  co

5.3     <p1> : if <bexp1> then <unc st1><block end1>
        is
        <p1> : if ¬ (<bexp1>) then goto <p1> 1 1; <unc st1>;
               <p1> 1 1 : <block end1>  co

5.4     <p1> : if <bexp1> then <for st1><block end1>
        is
        <p1> : if ¬ (<bexp1>) then goto <p1> 1 2; <for st1>
               <p1> 1 2 : <block end1>  co

5.5     <p1> : if <bexp1> then <unc st1> else <st1><block end1>
        is
        <p1> : if <bexp1> then begin <unc st1> ; goto <p1> 1 3 end;
               <st1> ; <p1> 1 3 : <block end1>  co

5.6     <p1> : if <bexp1> then goto <dexp1><block end1>
        is
        <p1> : goto if <bexp1> then <dexp1> else <p1> 1 4;
               <p1> 1 4 : <block end1>  co

5.7     <p1> : <unlabelled basic st1><block end1>
        is
        ( ⊀  <p1>  is  (  <unlabelled basic st1><p1> 2  co
             ⊀  <p1>  is  (
                ⊀  t <p1>  is  <unlabelled basic st1><p1> 4  ⊁  co
                <p1> a  ) ⊁  co
             <p1> a  )  ⊁  co
           <p1> a : <p1> k :  <block end1>  )  co

5.8     <p1> : <label1> : <st list1> end
        is
        (  label <label1><p1> 1  co
           ⊀  <p1>  is  (
                ⊀  <p1>  is  (  label <label1><p1> 3  co
                          ⊀ t <p1> is  t <p1> a ⊁  co
                   <p1> a  )  ⊁  co
             <p1> a  )  ⊁  co
           <p1> a : <st list1> end  )  co
```

```
5.9     <p1> : begin <decl list1><st list1> end <block end1>
        is
        <p1> : begin <decl list1><st list1> ; goto <p1> k 1 end;
               <p1> k 1 :  <block end1> co

5.10    begin <decl list><st list> ; goto <p> k 1 end
        in  <special block>  co

5.11    <p1> : <special block1><block end1>
        is
        ⌈ ≮  <p1>  is  (  <special block1> in  <decl block> co
                          first progr.p of block <p1> co
              ≮  <p1>  is  (
                         ≮  t <p1>  is  first progr.p of block <p1> ≯ co
                  <p1> a )  ≯ co
               <p1> a  )  ≯  co
           <p1> a : <block end1>  )  co

5.12    <bcs1><bc1>  im
        first progr.p of block <bcs1><as1>  is
        ≮  first progr.p of block <bcs1><as1>  is  <bcs1><bc1> a ≯ co

5.13    begin <block tail1> in  <decl block>
        is
        ⌈  begin  co <block tail1> ) co

5.14    <bcs1>  im  <block tail1>
        is
        ⌈ ≮  <bcs1> a  is  (
                ≮  <bcs1> a  is  (  begin  co
                        ≮  t <bcs1> a  is  t  <bcs1> a a ≯ co
                   <bcs1> a a )  ≯ co
              <bcs1> a a ) ≯ co
          <bcs1> a a : <block tail1>  )  co

5.15    <bcs1><as1> : end
        is
        ⌈≮ <bcs1><as1>  is  (  end co
                ≮ <bcs1><as1>  is  (
                        ≮ t <bcs1><as1>  is  end ≯ co
                   t <bcs1> a ) ≯ ) ≯ co
           <bcs1> a ) co
```

Value of begin and end.

6.1 <bn1> im begin is begin <bn1> co
6.2 <bn1> im end is end <bn1> co

6.3 begin <bcs1><dbcs1> is ⧼ <bcs1> b c <dbcs1> ⧽ co

6.4 <bcs1><bc1><dbcs> im
begin <bcs1><dbcs1> is ⧼ <bcs1> b <bc1><dbcs1> ⧽ co

6.5 end b c is
⧼ end b c is
⧼ <sequence of basic and aux term symbols> ⧽ ⧽ co

6.6 end <bcs1><bc><dbcs1> is ⧼ <bcs1><dbcs1> ⧽ co

Type declarations.

7.1 integer in <type> co
7.2 boolean in <type> co

7.3 own in <own> co

7.4 <id> in <id list> co
7.5 <id>,<id list> in <id list> co

7.6 <own><type><id list> in <type declaration> co

7.7 <own><type1><id1><bcs1><as> 1 is ⧼ <type1><id1><bcs1> ⧽ co

7.8 <specifier><id1><bcs1> im
<own><type><id1><bcs1><as> 1 is o co

7.9 <type declaration><p> 2 co

7.10 <type1><id1><p> 4 is <type1><id1> co

7.11 <bcs1><dbcs> im <type1><id1> is ⧼ <type1><id1><bcs1> ⧽ co

7.12 <bcs1><dbcs> im own <type1><id1><p1> 4
is
⧼ <type1><id1><bcs1> co
t <p1> is (own <type1><id1><p1> 4 <bcs1> co t <p1> a) ⧽ co

7.13 <bcs1><dbcs> im own <type1><id1><p1> 4 <bcs2>
is
(own <type1><id1><p1> 4 co ⧼ <id1><bcs1> is <id1><bcs2> ⧽) co

The value of a simple variable.

8.1 <bcs1><dbcs> im <id1> is <id1><bcs1> co

8.2 <id1><bcs1><bc> is <id1><bcs1> co

8.3 formal <id1><bcs1> actual <exp1> bn <bn1> im
<id1><bcs1>
is
(save bn <id1><bcs1> co ⊀ <bn1> ⊁ co result : va (<exp1>) co
reset bn <id1><bcs1> co result) co

8.4 <type><id1><bcs1> im <id1><bcs1> is o co

8.5 <bn1> im save bn <id1><bcs1>
is
⊀ reset bn <id1><bcs1> is ⊀ <bn1> ⊁ ⊁ co

8.6 result : <constant1> is ⊀ result is <constant1> ⊁ co

Array declarations.

```
9.1    <aexp> : <aexp>                   in  <bplist> co
9.2    <aexp> : <aexp> , <bplist>        in  <bplist> co

9.3    <decl aexp> : <decl aexp>                     in  <decl bplist> co
9.4    <decl aexp> : <decl aexp>,<decl bplist>       in  <decl bplist> co

9.5    <id>[<bplist>]          in  <array segment> co
9.6    <id>,<array segment>    in  <array segment> co

9.7    <array segment>                 in  <array list> co
9.8    <array segment>,<array list>    in  <array list> co

9.9    <own><type> array <array list>  in  <array declaration> co

9.10   <own><type1> array <id1>,<id list1>[<bplist1>]<p1> 1
       is
       (<type1> array <id1>      [<bplist1>]<p1> 1  co
        <type1> array <id list1>[<bplist1>]<p1> 1 ) co

9.11   <own><type1> array <id1>[<bplist1>]<bcs1><bc1><as> 1
       is
       (<type1> array <id1><bcs1><bc1> co ⊀ <bcs1> ⊁ co
        <bplist1> in  <decl bplist> co ⊀ <bcs1><bc1> ⊁ ) co

9.12   <type1> array <id1><bcs1>  is  ⊀ <type1> array <id1><bcs1> ⊁ co

9.13   <specifier><id1><bcs1> im <type> array <id1><bcs1>  is  o  co

9.14   <array declaration><p> 2  co

9.15   <own1><type1> array <id list1>[<bplist1>]<p1> 4
       is
       <own1><type1> array <id list1>[va ( <bplist1>)]<p1> 4  co

9.16   <bcs1><bc1><dbcs1>  im <aexp1> : <aexp2>
       is
       (  ⊀ <bcs1><dbcs1> ⊁ co
          bound pair : va ( <aexp1>) : va ( <aexp2>) co
          ⊀ <bcs1><bc1><dbcs1> ⊁ co bound pair )  co

9.17   bound pair : <int1> : <int2>
       is
       ⊀ bound pair is  <int1> : <int2> ⊁  co

9.18   <aexp1> :<aexp2> , <bplist1>
       is
       va ( <aexp1> : <aexp2> ) , va ( <bplist1> ) co
```

```
9.19   <int> : <int>                in  <int bplist> co
9.20   <int> : <int>,<int bplist>   in  <int bplist> co

9.21   <int bplist1>  is  ⊀<int bplist1> ⊁ co

9.22   <type1> array <id list1>[<int bplist1>]<p> 4
       is
       <type1> array <id list1>[<int bplist1>] co

9.23   <bcs1><dbcs>  im
       own <type1> array <id list1>[<int bplist1>]<p1> 4 <bcs2>
       is
       [⊀  t <p1> is
        ( own <type1> array <id list1>[<int bplist1>]<p1> 4 <bcs1> co
          t <p1> a ) ⊁  co
        own <type1> array <id list1>[<int bplist1>]<bcs2> ) co

9.24   <own1><type1> array <id1>,<id list1>[<int bplist1>]<bcs1>
       is
       [<own1><type1> array <id1>      [<int bplist1>]<bcs1> co
         <own1><type1> array <id list1>[<int bplist1>]<bcs1>  ] co

9.25   <bcs1><dbcs>  im
       <own> <type1> array <id1>[<int bplist1>]
       is
       ⊀<type1> array <id1><bcs1>[<int bplist1>] ⊁ co

9.26   <bcs1><dbcs>  im
       own <type1> array <id1>[<int bplist1>]<bcs2>
       is
       ⊀<type1> array <id1><bcs1>[<int bplist1>] co
         <sub exp list1> within bounds of <int bplist1> im
         <id1><bcs1>[<sub exp list1>]
         is
         <id1><bcs2>[<sub exp list1>] ⊁ co

9.27   <int3> within bounds of <int1> : <int2>
       is
       [ va ( <int3> - <int1> ) not negative co
         va ( <int2> - <int3> ) not negative ] co

9.28   <ui> not negative co

9.29   <int3>,<sub exp list1> within bounds of
       <int1> : <int2>,<bplist1>
       is
       [<int3> within bounds of <int1> : <int2> co
        <sub exp list1> within bounds of <bplist1> ) co
```

The value of a subscripted variable.

```
10.1   <int>                   in  <int list> co
10.2   <int>,<int list>        in  <int list> co

10.3   <sub exp1>,<sub exp list1>
       is
       va ( <sub exp1> ) , va ( <sub exp list1> ) co

10.4   <int list1>  is ≮ <int list1> ≯ co

10.5   <bcs1><dbcs>  im  <id1>[<sub exp list1>]
       is
       <id1><bcs1>[ va ( <sub exp list1> ) ] co

10.6   <id> b c [<sub exp list>]  is o  co

10.7   <id1><bcs1><bc>[<sub exp list1>]
       is
       <id1><bcs1>      [<sub exp list1>] co

10.8   formal <id1><bcs1> actual <id2> bn <bcs2><dbcs>  im
       <id1><bcs1>[<sub exp list1>]
       is
       <id2><bcs2>[<sub exp list1>]  co

10.9   <type> array <id1><bcs1>[<bplist>]  im
       <id1><bcs1>[<sub exp list>] is o  co
```

Switch declarations.

```
11.1   <dexp>                        in  <switch list> co
11.2   <dexp>,<switch list>          in  <switch list> co

11.3   <decl dexp>                          in  <decl switch list> co
11.4   <decl dexp>,<decl switch list>       in  <decl switch list> co

11.5   switch <id> := <switch list>  in  <switch declaration> co

11.6   switch <id1>:= <switch list><bcs1><as> 1  is
       ≮ switch <id1><bcs1> ≯ co

11.7   <specifier><id1><bcs1>  im
       switch <id1> := <switch list><bcs1><as> 1  is  o  co
```

```
11.8    switch <id> := <switch list1><p> 2  is
        <switch list1>  in  <decl switch list>  co

11.9    <bcs1><dbcs1>  im
        switch <id1>:= <switch list1><p> 4
        is
        ( ⊀ switch <id1><bcs1><dbcs1> ⊁ co
          store <id1><bcs1> := <switch list1> ) co

11.10   store <id1><bcs1> := <dexp1>
        is
        ⊀ <id1><bcs1>[1] eq <dexp1>  ⊁ co

11.11   store <id1><bcs1>:= <dexp1>,<switch list1>
        is
        ( store <id1><bcs1> := <dexp1> co
          store <id1><bcs1> [2] := <switch list1> ) co

11.12   store <id1><bcs1>[<ui1>] := <dexp1>
        is
        ⊀ <id1><bcs1>[<ui1] eq <dexp1> ⊁

11.13   store <id1><bcs1>[<ui1>] := <dexp1>,<switch list1>
        is
        ( store <id1><bcs1>[<ui1>] := <dexp1> co
          store <id1><bcs1>[ va ( <ui1> + 1 ) ]:= <switch list1> ) co
```

" Label declarations " .

```
12.1    label <label1><bcs1><as> 1
        is
        ⊀ label <label1><bcs1> ⊁ co

12.2    <specifier><label1><bcs1>  im
        label <label1><bcs1><as> 1  is  o  co

12.3    <bn1>  im  label  <label1><p1> 3
        is
        label <label1><bn1> 3 <p1>  co

12.4    <fgs1>  im label <label1><bn1> 3 <p1>
        is
        ⊀ label <label1><bn1><fgs1> eq t <p1> ⊁ co
```

Procedure declarations.

```
13.1    <type>                    in   <value specifier> co
13.2    <type> array              in   <value specifier> co
13.3    <type> procedure          in   <value specifier> co

13.4    <value specifier>         in   <specifier> co
13.5    label                     in   <specifier> co
13.6    switch                    in   <specifier> co
13.7    procedure                 in   <specifier> co

13.8    value <id list> ;  in  <value part>  co

13.9    <specifier><id list> ; ≤ spec part≥  in  <spec part> co

13.10   (<id list>)  in <formal par part> co

13.11   <type> procedure <id><formal par part> ;
        <value part> <spec part> <st>  in  <procedure declaration> co

13.12   <type1> procedure <id1><formal par part1>;
        <value part> <spec part> <st> <bcs1><as> 1
        is
        ⊀ <type1> procedure <id1><bcs1><formal par part1> ⊁ co

13.13   <specifier><id1><bcs1>  im
        <type> procedure <id1><formal par part> ;
        <value part> <spec part> <st> <bcs1><as> 1 is o co

13.14   procedure <id1> ; <st1><p1> 2
        is
        ( begin co formal <p1> k co
          begin integer dummy;<st1>;goto <p1> k end in <decl block> co
          first progr.p of proc.body <p1> co end ) co

13.15   <type> procedure <id1> ; <st1><p1> 2
        is
        ( begin integer dummy ; <st1> end in <decl block>  co
          first progr.p of proc.body <p1> ) co

13.16   procedure <id1>(<id list1>);
        <value part> <spec part> <st1><p1> 2
        is
        ( begin  co formal <id list1>,<p1> k co
          begin integer dummy;<st1>;goto <p1> k end in <decl block> co
          first progr.p of proc.body <p1> co end ) co
```

```
13.17   <type> procedure <id1>(<id list1>);
        <value part> <spec part> <st1><p1> 2
        is
        ( begin co formal <id list1> co
          begin integer dummy ; <st1> end in <decl block> co
          first progr.p of proc.body <p1> co end ) co

13.18   <bcs1> im formal <id1> is ≮ formal <id1><bcs1> ≯ co

13.19   formal <id1>,<id list1> is
        ( formal <id1> co formal <id list1> ) co

13.20   <bcs1><bc1> im first progr.p of proc.body <bcs1><as1> is
        ≮ first progr.p of proc.body <bcs1><as1>
          is  ≮ <bcs1><bc1> a ≯ ≯ co

13.21   <procedure declaration1><p1> 4
        is
        <procedure declaration1> : <p1> :
        va ( first progr.p of proc.body <p1> ) co

13.22   <bcs1><dbcs>  im
        procedure <id1> ; <st> : <p1> : <p2>
        is
        procedure <id1><bcs1> ( <p1> k) : <p2> co

13.23   <bcs1><dbcs>  im
        procedure <id1>(<id list1>);
        <value part1> <spec part1> <st> : <p1> : <p2>
        is
        procedure <id1><bcs1>(<id list1>,<p1> k) ;
        <value part1> <spec part1> : <p2> co

13.24   <bcs1><dbcs>  im
        <type1> procedure <id1><formal par part1> ;
        <value part1> <spec part1> <st> : <p> : <p1>
        is
        <type1> procedure <id1><bcs1><formal par part1> ;
        <value part1> <spec part1> : <p1> co

13.25   <value specifier><id>,<left formal list>
        in  <left formal list> co

13.26   ,<value specifier><id><right formal list>
        in  <right formal list> co

13.27   <left formal list> <value specifier><id><right formal list>
        in  <ext formal list> co

13.28   (<ext formal list>)  in  <ext formal par part> co
```

```
13.29   <type1> procedure <id1><bcs1><ext formal par part1> ;
        <spec part> : <p1>
        is
        ≮ <type1> procedure <id1><bcs1><ext formal par part1>:<p1> ≯ co

13.30   <type1> procedure <id1><bcs1>(<id list1>);
        value <id2>,<id list2>; <spec part1> : <p1>
        is
        <type1> procedure <id1><bcs1>(<id list1>);
        value <id2> ; value <id list2> ; <spec part1> : <p1> co

13.31   <type1> procedure <id1><bcs1>
        (<left formal list1><id2><right formal list1>);
        value <id2>; <value part1> <spec part1>
        <value specifier1><left formal list2><id2><right formal list2>;
        <spec part2> : <p1>
        is
        <type1> procedure <id1><bcs1>
        (<left formal list1><value specifier1><id2><right formal list1>);
        <value part1> <spec part1><value specifier1>
        <left formal list2><id2><right formal list2>;
        <spec part2> :<p1> co
```

Assignment statements.

14.1 <int var>:= in <int left part> co
14.2 <int proc id>:= in <int left part> co

14.3 <decl int var>:= in <decl int left part> co
14.4 <decl int proc id>:= in <decl int left part> co

14.5 <boolean var>:= in <boolean left part> co
14.6 <boolean proc id>:= in <boolean left part> co

14.7 <decl boolean var>:= in <decl boolean left part> co
14.8 <decl boolean proc id>:= in <decl boolean left part> co

14.9 <int left part><int left part list>
in <int left part list> co

14.10 <boolean left part><boolean left part list>
in <boolean left part list> co

14.11 <decl int left part><decl int left part list>
in <decl int left part list> co

14.12 <decl boolean left part><decl boolean left part list>
in <decl boolean left part list> co

14.13 <int left part list><aexp> in <ass st> co
14.14 <boolean left part list><bexp> in <ass st> co

14.15 <decl int left part list><decl aexp> in <decl ass st> co
14.16 <decl boolean left part list><decl bexp> in <decl ass st> co

14.17 <ass st1><p1> 2 is <ass st1> in <decl ass st> co

14.18 <ass st1><p1> 4 is (<ass st1> co t <p1> a) co

14.19 <type><id><bcs> in <ext left part> co

14.20 <type> array <id><bcs>[<sub exp list>] in <ext left part> co

14.21 <int left part> in <ext left part list> co
14.22 <boolean left part> in <ext left part list> co
14.23 <ext left part> in <ext left part list> co

14.24 <ext left part list><ext left part list>
in <ext left part list> co

```
14.25   <ext left part1><ext left part list1><exp1>
        is
        <ext left part1><ext left part list1> va ( <exp1> ) co

14.26   <ext left part1><ext left part list1><constant1>
        is
        (<ext left part1><constant1> co
         <ext left part list1><constant1> ) co

14.27   <ext left part><constant>  is o co

14.28   <bcs1><dbcs>  im  <id1>:= <ext left part list1><exp1>
        is
        <id1><bcs1>:= <ext left part list1><exp1> co

14.29   <id> b c := <ext left part list><exp>  is o co

14.30   <id1><bcs1><bc>:= <ext left part list1><exp1>
        is
        <id1><bcs1>     := <ext left part list1><exp1> co

14.31   formal <id1><bcs1> actual <id2> bn <bcs2><dbcs>  im
        <id1><bcs1>:= <ext left part list1><exp1>
        is
        <id2><bcs2>:= <ext left part list1><exp1> co

14.32   formal <id1><bcs1> actual <id2>[<sub exp list1>]
        bn <bcs2><dbcs1> im
        <id1><bcs1>:= <ext left part list1><exp1>
        is
        ( save bn <id1><bcs1> co ⊀ <bcs2><dbcs1> ⊁ co
          subscript list : va ( <sub exp list1> ) co
          reset bn <id1><bcs1> co <id2><bcs2>[ va ( subscript list )]:=
          <ext left part list1><exp1> ) co

14.33   subscript list : <int list1>
        is
        ⊀ subscript list is <int list1> ⊁ co

14.34   <type1><id1> p <bcs1> im
        <id1><bcs1>:= <ext left part list1><exp1>
        is
        <ext left part list1><type1><id1> p <bcs1>:= <exp1> co

14.35   <type1><id1><bcs1> im
        <id1><bcs1>:= <ext left part list1><exp1>
        is
        <ext left part list1><type1><id1><bcs1>:= <exp1> co
```

```
14.36   <bcs1><dbcs> im
        <id1>[<sub exp list1>]:= <ext left part list><exp1>
        is
        <id1><bcs1>[ va ( <sub exp list1> )]:=
        <ext left part list1><exp1> co

14.37   <id> b c [<sub exp list>]:= <ext left part list><exp> is o co

14.38   <id1><bcs1><bc>[<sub exp list1>]:= <ext left part list1><exp1>
        is
        <id1><bcs1>[<sub exp list1>]:= <ext left part list1><exp1> co

14.39   formal <id1><bcs1> actual <id2> bn <bcs2><dbcs> im
        <id1><bcs1>[<sub exp list1>]:= <ext left part list1><exp1>
        is
        <id2><bcs2>[<sub exp list1>]:= <ext left part list1><exp1> co

14.40   <type1> array <id1><bcs1>[<int bplist1>] im
        <id1><bcs1>[<sub exp list1>]:= <ext left part list1><exp1>
        is
        ( <sub exp list1> within bounds of <int bplist1> co
          <ext left part list1><type1> array
          <id1><bcs1>[<sub exp list1>]:= <exp1> ) co

14.41   integer <id1><bcs1> := <int1> is ⊀ <id1><bcs1> is <int1> ⊁ co

14.42   boolean <id1><bcs1>:= <logical value1>
        is
        ⊀ <id1><bcs1> is <logical value1> ⊁ co

14.43   integer array <id1><bcs1>[<sub exp list1>]:= <int1>
        is
        ⊀ <id1><bcs1>[<sub exp list1>] is <int1> ⊁ co

14.44   boolean array <id1><bcs1>[<sub exp list1>]:= <logical value1>
        is
        ⊀ <id1><bcs1>[<subexp list1>] is <logical value1> ⊁ co
```

Goto statements.

```
15.1    goto <dexp1><p1> 2  is <dexp1> in <decl dexp> co

15.2    <bn1> im
        goto <dexp1><p1> 4 is goto <dexp1><p1><bn1> co

15.3    goto (<dexp1>)<p1><bn1> is goto <dexp1><p1><bn1> co

15.4    goto if <bexp1> then <sdexp1> else <dexp1><p1><bn1>
        is
        goto if va ( <bexp1>) then <sdexp1> else <dexp1><p1><bn1> co

15.5    goto if true then <sdexp1> else <dexp><p1><bn1>
        is
        goto <sdexp1><p1><bn1> co

15.6    goto if false then <sdexp> else <dexp1><p1><bn1>
        is
        goto <dexp1><p1><bn1> co

15.7    <fgs1>  im
        goto <label1><p1><bn1> is goto <label1><fgs1><p1><bn1> co

15.8    <bcs1><dbcs>  im goto <label1><fgs1><p1><bn1>
        is
        goto <label1><bcs1><fgs1><p1><bn1> co

15.9    goto <label> b c <fgs><p><bn>  is  o  co

15.10   goto <label1><bcs1><bc><fgs1><p1><bn1>
        is
        goto <label1><bcs1><fgs1><p1><bn1> co

15.11   formal <id1><bcs1> actual <dexp1> bn <bn2> im
        goto <id1><bcs1><fgs><p1><bn1>
        is
        ⌈ ≮ <bn2> ≯ co goto <dexp1><p1><bn1> ) co

15.12   label <label1><bcs1><dbcs1><fgs1> eq t <p1> im
        goto <label1><bcs1><fgs1><fgs><p><bn>
        is
        ⌈ ≮ <bcs1><dbcs1> ≯ co ≮ <fgs1> ≯ co t <p1> ) co

15.13   <bcs1><dbcs>  im goto <id1>[<sub exp1>]<p1><bn1>
        is
        goto <id1><bcs1>[ va ( <sub exp1>)]<p1><bn1> co
```

```
15.14   goto <id> b c [<sub exp>]<p><bn> is o co

15.15   goto <id1><bcs1><bc>[<sub exp1>]<p1><bn1>
        is
        goto <id1><bcs1>[<sub exp1>]<p1><bn1> co

15.16   formal <id1><bcs1> actual <id2> bn <bcs2><dbcs> im
        goto <id1><bcs1>[<sub exp1>]<p1><bn1>
        is
        goto <id2><bcs2>[<sub exp1>]<p1><bn1> co

15.17   switch <id1><bcs1><dbcs1> im
        goto <id1><bcs1>[<sub exp1>]<p1><bn1>
        is
        ( ≮ <bcs1><dbcs1> ≯ co go <id1><bcs1>[<sub exp1>]<p1><bn1> ) co

15.18   go <id><bcs>[<sub exp>]<p1><bn1>
        is
        ( ≮ <bn1> ≯ co t <p1> a ) co

15.19   <id1><bcs1>[<sub exp1>] eq <dexp1> im
        go <id1><bcs1>[<sub exp1>]<p1><bn1>
        is
        goto <dexp1><p1><bn1> co
```

For statements.

```
16.1   <aexp>                                  in  <for list el> co
16.2   <aexp> while <bexp>                     in  <for list el> co
16.3   <aexp> step <aexp> until <aexp>         in  <for list el> co

16.4   <for list el>                   in  <for list> co
16.5   <for list el>,<for list>        in  <for list> co

16.6   for <int var> := <for list> do <st>  in  <for st> co
16.7   <label> : <for st>                   in  <for st> co

16.8   f <fs>  in  <fs> co

16.9   <fs> g <fgs>  in  <fgs> co

16.10  <fgs1>  im  forbegin  is  forbegin <fgs1> co

16.11  forbegin <fgs1>  is ⦃ <fgs1> f g ⦄ co

16.12  <fgs1><fs1> g  im
       forbegin <fgs1>  is  ⦃ <fgs1><fs1> f g ⦄ co

16.13  <p1> : for <int var1> := <for list1> do <st1><block end1>
       is
       [⦃ <p1> is (
               ⦃ <p1> is ( forbegin co
                      ⦃ t <p1> is t <p1> a ⦄ co
                 <p1> a ) ⦄ co
           <p1> a ) ⦄ co
         <p1> a : <int var1> := <for list1>;
         <p1> m 1 : <st1> ; goto special label <p1> ;
         <p1> m 2 : forend ( <int var1>) ;
         <block end1> ) co

16.14  <p1> : <int var1>:= <aexp1>,<for list1>;<p2> m 1 : <block end1>
       is
       [⦃ <p1> is (
          ⦃ <p1> is (
              ⦃ t <p1> is ( ⦃ special label <p2> : <p1> m 3 ⦄ co
                t <p1> a ) ⦄ co
             <p1> a ) ⦄ co
           <p1> a ] ⦄ co
         <p1> a : <int var1>:= <aexp1>; goto <p2> m 1 ;
         <p1> m 3 : <int var1>:= <for list1>;
         <p2> m 1 : <block end1> ) co
```

```
16.15  <p1> : <int var1>:= <aexp1> while <bexp1>,<for list1>;
              <p2> m 1 : <block end1>
       is
       [⊀ <p1> is (
           ⊀ <p1> is (
               ⊀ t <p1> is ( ⊀ special label <p2> : <p1> m 4 ⊁ co
                 t <p1> a ) ⊁ co
             <p1> a ) ⊁ co
           <p1> a ) ⊁ co
         <p1> a : <p1> m 4 : <int var1>:= <aexp1>;
         if <bexp1> then goto <p2> m 1 ; <int var1>:= <for list1>;
         <p2> m 1 : <block end1> ) co

16.16  <p1> : <int var1> := <aexp1> step <aexp2> until <aexp3>,
              <for list1>; <p2> m 1 : <block end1>
       is
       [⊀ <p1> is (
           ⊀ <p1> is (
               ⊀ t <p1> is ( ⊀ special label <p2> : <p1> m 5 ⊁ co
                 t <p1> a ) ⊁ co
              <p1> a ) ⊁ co
          <p1> a ) ⊁ co
         <p1> a : <int var1>:= <aexp1>;
         <p1> m 6 : goto if (<int var1>—<aexp3>) × sign (<aexp2>) > 0
                         then <p1> m 7 else <p2> m 1 ;
         <p1> m 5 : <int var1> := <int var1> + <aexp1>; goto <p1> m 6 ;
         <p1> m 7 : <int var1> := <for list1>;
         <p2> m 1 : <block end1> ) co

16.17  <p1> : <int var1>:= <aexp1>;<p2> m 1 : <block end1>
       is
       [⊀ <p1> is ( <int var1>:= <aexp1> in <decl ass st> co
           ⊀ <p1> is (
               ⊀ t <p1> is ( ⊀ special label <p2> : <p2> m 2 ⊁ co
                               <int var1> := <aexp1> co
                 t <p1> a ) ⊁ co
             <p1> a ) ⊁ co
           <p1> a ] ⊁ co
         <p1> a : <p2> m 1 : <block end1> ) co

16.18  <p1> : <int var1>:= <aexp1> while <bexp1> ;
              <p2> m 1 : <block end1>
       is
       [⊀ <p1> is (
           ⊀ <p1> is (
               ⊀ t <p1> is ( ⊀ special label <p2> : <p1> m 8 ⊁ co
                 t <p1> a ) ⊁ co
             <p1> a  ) ⊁ co
           <p1> a ) ⊁ co
         <p1> a : <p1> m 8 : <int var1> := <aexp1>;
         goto if <bexp1> then <p2> m 1 else <p2> m 2;
         <p2> m 1 : <block end1> ) co
```

```
16.19   <p1> : <int var1>:= <aexp1> step <aexp2> until <aexp3> ;
               <p2> m 1 : <block end1>
        is
        [⊀ <p1> is (
            ⊀ <p1> is (
                ⊀ t <p1> is ( ⊀ special label <p2> : <p1> m 9 ⊁ co
                  t <p1> a ) ⊁ co
              <p1> a ) ⊁ co
            <p1> a ] ⊁ co
          <p1> a : <int var1> := <aexp1> ;
          <p1> m 10 : goto if (<int var1>-<aexp3>) × sign(<aexp2>) > 0
                           then <p2> m 2 else <p2> m 1 ;
          <p1> m 9 : <int var1>:= <int var1> + <aexp2> ;
                     goto <p1> m 10;
          <p2> m 1 : <block end1> ) co

16.20   <p1> : goto special label <p2><block end1>
        is
        [⊀ <p1> is (
            ⊀ <p1> is (
                ⊀ t <p1> is goto special label <p2> ⊁ co
              <p1> a ) ⊁ co
            <p1> a ] ⊁ co
          <p1> a : <block end1> ) co

16.21   special label <p1> : <p2> m <ui1> im
        goto special label <p1> is goto <p2> m <ui1> co

16.22   <p1> : forend (<int var1>) <block end1>
        is
        [⊀ <p1> is (
            ⊀ <p1> is (
              ⊀ t <p1> is ( forend (<int var1>) co
                t <p1> a ) ⊁ co
              <p1> a ) ⊁ co
            <p1> a ] ⊁ co
          <p1> a : <block end1> ) co

16.23   <fgs1><fs> g im  forend (<int var1>)
        is
        [⊀ <fgs1> ⊁ co forend <int var1> ) co

16.24   <bcs1><dbcs> im
        forend <id1>  is forend <id1><bcs1> co

16.25   forend <id1><bcs1><bc> is forend <id1><bcs1> co

16.26   formal <id1><bcs1> actual <id2> bn <bcs2><dbcs>  im
        forend <id1><bcs1> is forend <id2><bcs2> co
```

```
16.27   formal <id1><bcs1> actual <id2>[<sub exp list1>]
        bn <bcs2><dbcs1>  im
        forend <id1><bcs1>
        is
        ( save bn <id1><bcs1> co ⊀ <bcs2><dbcs1>  ⊁ co
          subscript list : va ( <sub exp list1> ) co
          reset bn <id1><bcs1> co
          forend <id2><bcs2>[ va ( subscript list ) ] ) co

16.28   integer <id1><bcs1>  im forend <id1><bcs1>
        is
        ⊀ <id1><bcs1> is o ⊁ co

16.29   <bcs1><dbcs> im
        forend <id1>[<sub exp list1>]
        is
        forend <id1><bcs1>[ va ( <sub exp list1> ) ] co

16.30   forend <id1><bcs1><bc>[<sub exp list1>]
        is
        forend <id1><bcs1>[<sub exp list1>] co

16.31   formal <id1><bcs1> actual <id2> bn <bcs2><dbcs>  im
        forend <id1><bcs1>[<sub exp list1>]
        is
        forend <id2><bcs2>[<sub exp list1>] co

16.32   integer array <id1><bcs1>[<int bplist>]
        im
        forend <id1><bcs1>[<sub exp list1>]
        is
        ⊀ <id1><bcs1>[<sub exp list1>] is o ⊁ co
```

Procedure statements and function designators.

17.1 <id> in <proc id> co
17.2 <id> in <int proc id> co
17.3 <id> in <boolean proc id> co

17.4 <proc id><act par part> in <proc st> co
17.5 <int proc id><act par part> in <int funct des> co
17.6 <boolean proc id><act par part> in <boolean funct des> co

17.7 <exp> in <act par> co
17.8 <int array id> in <act par> co
17.9 <boolean array id> in <act par> co
17.10 <switch id> in <act par> co
17.11 <proc id> in <act par> co
17.12 <int proc id> in <act par> co
17.13 <boolean proc id> in <act par> co

17.14 <act par> in <act par list> co
17.15 <act par>,<act par list> in <act par list> co

17.16 (<act par list>) in <act par part> co

17.17 <decl exp> in <decl act par> co
17.18 <decl int array id> in <decl act par> co
17.19 <decl boolean array id> in <decl act par> co
17.20 <decl switch id> in <decl act par> co
17.21 <decl proc id> in <decl act par> co
17.22 <decl int proc id> in <decl act par> co
17.23 <decl boolean proc id> in <decl act par> co

17.24 <decl act par> in <decl act par list> co
17.25 <decl act par>,<decl act par list> in <decl act par list> co

17.26 (<decl act par list>) in <decl act par part> co

17.27 <bcs1> im <id1> in <decl proc id> is
<id1><bcs1> in <decl proc id> co
17.28 <bcs1> im <id1> in <decl int proc id> is
<id1><bcs1> in <decl int proc id> co
17.29 <bcs1> im <id1> in <decl boolean proc id> is
<id1><bcs1> in <decl boolean proc id> co

17.30 <id> b c in <decl proc id> is o co
17.31 <id> b c in <decl int proc id> is o co
17.32 <id> b c in <decl boolean proc id> is o co

17.33 <id1><bcs1><bc> in <decl proc id> is
<id1><bcs1> in <decl proc id> co

17.34 <id1><bcs1><bc> in <decl int proc id> is
<id1><bcs1> in <decl int proc id> co

17.35 <id1><bcs1><bc> in <decl boolean proc id> is
<id1><bcs1> in <decl boolean proc id> co

17.36 formal <id1><bcs1> im <id1><bcs1> in <decl proc id> co
17.37 formal <id1><bcs1> im <id1><bcs1> in <decl int proc id> co
17.38 formal <id1><bcs1> im <id1><bcs1> in <decl boolean proc id> co

17.39 <type> procedure <id1><bcs1><formal par part> im
<id1><bcs1> in <decl proc id> co

17.40 integer procedure <id1><bcs1><formal par part1> im
<id1><bcs1> in <decl int proc id> co

17.41 boolean procedure <id1><bcs1><formal par part1> im
<id1><bcs1> in <decl boolean proc id> co

17.42 <bcs1> im
<id1><decl act par part1> in <decl proc st> is
<id1><bcs1><decl act par part1> in <decl proc st> co

17.43 <bcs1> im
<id1><decl act par part1> in <decl int funct des> is
<id1><bcs1><decl act par part1> in <decl int funct des> co

17.44 <bcs1> im
<id1><decl act par part1> in <decl boolean funct des> is
<id1><bcs1><decl act par part1> in <decl boolean funct des> co

17.45 <id> b c <act par part> in <decl proc st> is o co
17.46 <id> b c <act par part> in <decl int funct des> is o co
17.47 <id> b c <act par part> in <decl boolean funct des> is o co

17.48 <id1><bcs1><bc><act par part1> in <decl proc st> is
<id1><bcs1> <act par part1> in <decl proc st> co

17.49 <id1><bcs1><bc><act par part1> in <decl int funct des> is
<id1><bcs1> <act par part1> in <decl int funct des> co

17.50 <id1><bcs1><bc><act par part1> in <decl boolean funct des> is
<id1><bcs1> <act par part1> in <decl boolean funct des> co

17.51 formal <id1><bcs1> im
<id1><bcs1><act par part> in <decl proc st> co

17.52 formal <id1><bcs1> im
<id1><bcs1><act par part> in <decl int funct des> co

17.53 formal <id1><bcs1> im
<id1><bcs1><act par part> in <decl boolean funct des> co

17.54 <type> procedure <id1><bcs1> im
<id1><bcs1> in <decl proc st> co

17.55 integer procedure <id1><bcs1> im
<id1><bcs1> in <decl int funct des> co

17.56 boolean procedure <id1><bcs1> im
<id1><bcs1> in <decl boolean funct des> co

17.57 <type> procedure <id1><bcs1>(<id list1>) im
<id1><bcs1>(<act par list1>) in <decl proc st>
is
<id list1> equal length <act par list1> co

17.58 integer procedure <id1><bcs1>(<id list1>) im
<id1><bcs1>(<act par list1>) in <decl int funct des>
is
<id list1> equal length <act par list1> co

17.59 boolean procedure <id1><bcs1>(<id list1>) im
<id1><bcs1>(<act par list1>) in <decl boolean funct des>
is
<id list1> equal length <act par list1> co

17.60 <id> equal length <act par> co

17.61 <id>,<id list1> equal length <act par>,<act par list1>
is
<id list1> equal length <act par list1> co

17.62 <proc st1><p> 2 is <proc st1> in <decl proc st> co

17.63 <proc st1><p1> 4 is <proc st1> : <p1> co

17.64 <bcs1><dbcs> im <id1><act par part1> : <p1>
is
<id1><bcs1><act par part1> : <p1> co

17.65 <bcs1><dbcs> im <id1>(<act par list1>)
is
<id1><bcs1>(<act par list1>) co

17.66 <id> b c <act par part> : <p> is o co

17.67 <id> b c (<act par list>) is o co

```
17.68  <id1><bcs1><bc><act par part1> : <p1> is
       <id1><bcs1>    <act par part1> : <p1> co

17.69  <id1><bcs1><bc>(<act par list1>) is
       <id1><bcs1>    (<act par list1>) co

17.70  formal <id1><bcs1> actual <id2> bn <bcs2><dbcs> im
       <id1><bcs1><act par part1> : <p1>
       is
       <id2><bcs2><act par part1> : <p1> co

17.71  formal <id1><bcs1> actual <id2> bn <bcs2><dbcs> im
       <id1><bcs1>(<act par list1>)
       is
       <id2><bcs2>(<act par list1>) co

17.72  procedure <id1><bcs1>(<p3> k) : <p1> im
       <id1><bcs1> : <p2>
       is
       ( enter procedure <bcs1> co
         (<p3> k) substitute (<p2> k) co <p1> ) co

17.73  procedure <id1><bcs1>(<ext formal list1>) : <p1> im
       <id1><bcs1>(<act par list1>) : <p2>
       is
       ( enter procedure <bcs1> co
         (<ext formal list1>) substitute (<act par list1>,<p2> k) co
         <p1> ) co

17.74  <type1> procedure <id1><bcs1><ext formal par part1> : <p1> im
       <id1><bcs1><act par part1>
       is
       ( enter procedure <bcs1> co begin co <type1><id1> p co
         <ext formal par part1> substitute <act par part1> co
         <p1> co function value : va ( <id1> p ) co
         exit procedure co function value ) co

17.75  integer procedure <id1><bcs1><ext formal par part1> :<p> im
       <id1><bcs1><act par part1> : <p1>
       is
       (  dummy 1 := <id1><act par part1> co t <p1> a ) co

17.76  boolean procedure <id1><bcs1><ext formal par part1> : <p> im
       <id1><bcs1><act par part1> : <p1>
       is
       ( dummy 2 := <id1><act par part1> co t <p1> a ) co

17.77  <bn1> im
       enter procedure <bcs1> is ≮ <bcs1> d <bn1> ≯ co
```

17.78 <bcs> d <bn1> im
exit procedure is ≮ <bn1> ≯ co

17.79 function value : <constant1>
is
≮ function value is <constant1> ≯ co

17.80 ,<act par><right act par list> in <right act par list> co

17.81 (<type1><id1><right formal list1>) substitute
(<act par1><right act par list1>)
is
(begin co <type1><id1> co <type1><id1> becomes <act par1> co
(<right formal list1>) substitute (<right act par list1>)) co

17.82 <bcs1> d <bn1> im <type1><id1> becomes <act par1>
is
(≮ <bn1> ≯ co <type1><id1><bcs1>:= <act par1> co
≮ <bcs1> d <bn1> ≯) co

17.83 (<type1> procedure <ext formal list1>) substitute
(<act par list1>)
is
(<type1><ext formal list1>) substitute (<act par list1>) co

17.84 (<type1> array <id1><right formal list1>) substitute
(<id2><right act par list1>)
is
(begin co <type1> array <id1> actual <id2> co
(<right formal list1>) substitute (<right act par list1>)) co

17.85 <bcs1> d <bcs2><dbcs> im
<type1> array <id1> actual <id2>
is
<type1> array <id1><bcs1> actual <id2><bcs2> co

17.86 <type> array <id><bcs> actual <id> b c is o co

17.87 <type1> array <id1><bcs1> actual <id2><bcs2><bc>
is
<type1> array <id1><bcs1> actual <id2><bcs2> co

17.88 formal <id2><bcs2> actual <id3> bn <bcs3><dbcs> im
<type1> array <id1><bcs1> actual <id2><bcs2>
is
<type1> array <id1><bcs1> actual <id3><bcs3> co

17.89 <type1> array <id1><bcs1>[<int bplist1>] im
<type1> array <id2><bcs2> actual <id1><bcs1>
is
(≮ <type1> array <id2><bcs2>[<int bplist1>] ≯ co
<id2><bcs2>[<int bplist1>] assign <id1><bcs1>) co

17.90 <int>,<left int list> in <left int list> co

17.91 <id1><bcs1>[<left int list1><int1> :<int2>] assign <id2><bcs2>
is
(<id1><bcs1>[<left int list1><int1>] becomes
va (<id2><bcs2>[<left int list1><int1>]) co
<id1><bcs1>[<left int list1> va (<int1> + 1) : <int2>]
assign <id2><bcs2>) co

17.92 <id1><bcs1>[<left int list1><int1> : <int2>,<bplist1>]
assign <id2><bcs2>
is
(<id1><bcs1>[<left int list1><int1>,<bplist1>]
assign <id2><bcs2> co <id1><bcs1>
[<left int list1> va (<int1> + 1) : <int2>,<bplist1>]
assign <id2><bcs2>) co

17.93 <int1> equal <int2> im
<id1><bcs1>[<left int list1><int1> : <int2>,<bplist1>]
assign <id2><bcs2>
is
<id1><bcs1>[<left int list1><int1>,<bplist1>]
assign <id2><bcs2> co

17.94 <int1> equal <int2> im
<id1><bcs1>[<left int list1><int1> : <int2>]
assign <id2><bcs2>
is
<id1><bcs1>[<left int list1><int1>] becomes
va (<id2><bcs2>[<left int list1><int1>]) co

17.95 <id1><bcs1>[<int list1>] becomes <constant1>
is
(<id1><bcs1>[<int list1>] is <constant1>) co

17.96 <int1> equal <int2> is va (<int1> – <int2>) equal zero co

17.97 0 equal zero co

```
17.98   (<id1>,<ext formal list1>) substitute
        (<act par1>,<act par list1>)
        is
        (<ext formal list1>,<id1>) substitute
        (<act par list1>,<act par1>) co

17.99   ,<id><right id list> in <right id list> co

17.100  (<id1><right id list1>) substitute
        (<act par1><right act par list1>)
        is
        ( begin co <id1> actual <act par1> co
         (<right id list1>) substitute (<right act par list1>) ) co

17.101  <bcs1> d <bn1> im <id1> actual <act par1>
        is
        { formal <id1><bcs1> actual <act par1> bn <bn1> } co

17.102  (,<ext formal list1>) substitute (,<act par list1>)
        is
        (<ext formal list1>) substitute (<act par list1>) co

17.103  ( ) substitute ( ) co

17.104  substitute co
```

Variables.

```
18.1    <aexp> in <exp> co
18.2    <bexp> in <exp> co
18.3    <dexp> in <exp> co

18.4    <decl aexp> in <decl exp> co
18.5    <decl bexp> in <decl exp> co
18.6    <decl dexp> in <decl exp> co

18.7    <id> in <int var id> co
18.8    <id> in <boolean var id> co
18.9    <id> in <int array id> co
18.10   <id> in <boolean array id> co

18.11   <bcs1> im <id1>         in <decl int var id> is
                  <id1><bcs1>   in <decl int var id> co
18.12   <bcs1> im <id1>         in <decl boolean var id> is
                  <id1><bcs1>   in <decl boolean var id> co
18.13   <bcs1> im <id1>         in <decl int array id> is
                  <id1><bcs1>   in <decl int array id> co
18.14   <bcs1> im <id1>         in <decl boolean array id> is
                  <id1><bcs1>   in <decl boolean array id> co

18.15   <id> b c in <decl int var id>         is o co
18.16   <id> b c in <decl boolean var id>     is o co
18.17   <id> b c in <decl int array id>       is o co
18.18   <id> b c in <decl boolean array id>   is o co

18.19   <id1><bcs1><bc>   in <decl int var id> is
        <id1><bcs1>       in <decl int var id> co
18.20   <id1><bcs1><bc>   in <decl boolean var id> is
        <id1><bcs1>       in <decl boolean var id> co
18.21   <id1><bcs1><bc>   in <decl int array id> is
        <id1><bcs1>       in <decl int array id> co
18.22   <id1><bcs1><bc>   in <decl boolean array id> is
        <id1><bcs1>       in <decl boolean array id> co

18.23   formal <id1><bcs1> im <id1><bcs1> in <decl int var id> co
18.24   formal <id1><bcs1> im <id1><bcs1> in <decl boolean var id> co
18.25   formal <id1><bcs1> im <id1><bcs1> in <decl int array id> co
18.26   formal <id1><bcs1> im <id1><bcs1> in <decl boolean array id> co

18.27   integer <id1><bcs1> im <id1><bcs1> in <decl int var id> co
18.28   boolean <id1><bcs1> im <id1><bcs1> in <decl boolean var id> co

18.29   integer array <id1><bcs1> im
        <id1><bcs1> in <decl int array id> co
18.30   boolean array <id1><bcs1> im
        <id1><bcs1> in <decl boolean array id> co
```

18.31 <int var id> in <int var> co

18.32 <boolean var id> in <boolean var> co

18.33 <decl int var id> in <decl int var> co

18.34 <decl boolean var id> in <decl boolean var> co

18.35 <aexp> in <sub exp> co

18.36 <decl aexp> in <decl sub exp> co

18.37 <sub exp> in <sub exp list> co

18.38 <sub exp>,<sub exp list> in <sub exp list> co

18.39 <decl sub exp> in <decl sub exp list> co

18.40 <decl sub exp>,<decl sub exp list> in <decl sub exp list> co

18.41 <int array id>[<sub exp list>] in <int var> co

18.42 <boolean array id>[<sub exp list>] in <boolean var> co

18.43 <decl int array id>[<decl sub exp list>]
in <decl int var> co

18.44 <decl boolean array id>[<decl sub exp list>]
in <decl boolean var> co

Syntax of arithmetic expressions.

19.1	+	in	<pm>	co
19.2	−	in	<pm>	co
19.3	×	in	<mult op>	co
19.4	:	in	<mult op>	co
19.5	<ui>	in	<primary>	co
19.6	<int var>	in	<primary>	co
19.7	<int funct des>	in	<primary>	co
19.8	(<aexp>)	in	<primary>	co
19.9	<ui>	in	<decl primary>	co
19.10	<decl int var>	in	<decl primary>	co
19.11	<decl int funct des>	in	<decl primary>	co
19.12	(<decl aexp>)	in	<decl primary>	co
19.13	<primary>	in	<factor>	co
19.14	<factor> ↑ <primary>	in	<factor>	co
19.15	<decl primary>	in	<decl factor>	co
19.16	<decl factor> ↑ <decl primary>	in	<decl factor>	co
19.17	<factor>	in	<term>	co
19.18	<term><mult op><factor>	in	<term>	co
19.19	<decl factor>	in	<decl term>	co
19.20	<decl term><mult op><decl factor>	in	<decl term>	co
19.21	<term>	in	<saexp>	co
19.22	<pm><term>	in	<saexp>	co
19.23	<saexp><pm><term>	in	<saexp>	co
19.24	<decl term>	in	<decl saexp>	co
19.25	<pm><decl term>	in	<decl saexp>	co
19.26	<decl saexp><pm><decl term>	in	<decl saexp>	co
19.27	<saexp>	in	<aexp>	co
19.28	<decl saexp>	in	<decl aexp>	co
19.29	if <bexp> then <saexp> else <aexp>	in	<aexp>	co
19.30	if <decl bexp> then <decl saexp> else <decl aexp>	in	<decl aexp>	co

Syntax of boolean expressions.

20.1 < in <rel op> co
20.2 > in <rel op> co
20.3 ≤ in <rel op> co
20.4 ≥ in <rel op> co
20.5 = in <rel op> co
20.6 ≠ in <rel op> co

20.7 <logical value> in <bprimary> co
20.8 <boolean var> in <bprimary> co
20.9 <saexp><rel op><saexp> in <bprimary> co
20.10 (<bexp>) in <bprimary> co
20.11 <boolean funct des> in <bprimary> co

20.12 <logical value> in <decl bprimary> co
20.13 <decl boolean var> in <decl bprimary> co
20.14 <decl saexp><rel op><decl saexp> in <decl bprimary> co
20.15 (<decl bexp>) in <decl bprimary> co
20.16 <decl boolean funct des> in <decl bprimary> co

20.17 <bprimary> in <bsecondary> co
20.18 ¬ <bprimary> in <bsecondary> co

20.19 <decl bprimary> in <decl bsecondary> co
20.20 ¬ <decl bprimary> in <decl bsecondary> co

20.21 <bsecondary> in <bfactor> co
20.22 <bfactor> ∧ <bsecondary> in <bfactor> co

20.23 <decl bsecondary> in <decl bfactor> co
20.24 <decl bfactor> ∧ <decl bsecondary> in <decl bfactor> co

20.25 <bfactor> in <bterm> co
20.26 <bterm> ∨ <bfactor> in <bterm> co

20.27 <decl bfactor> in <decl bterm> co
20.28 <decl bterm> ∨ <decl bfactor> in <decl bterm> co

20.29 <bterm> in <implication> co
20.30 <implication> ⊃ <bterm> in <implication> co

20.31 <decl bterm> in <decl implication> co
20.32 <decl implication> ⊃ <decl bterm> in <decl implication> co

20.33 <implication> in <sbexp> co
20.34 <sbexp> = <implication> in <sbexp> co

20.35 <decl implication> in <decl sbexp> co
20.36 <decl sbexp> = <decl implication> in <decl sbexp> co

20.37 <sbexp> in <bexp> co
20.38 if <bexp> then <sbexp> else <bexp> in <bexp> co

20.39 <decl sbexp> in <decl bexp> co

20.40 if <decl bexp> then <decl sbexp> else <decl bexp>
in <decl bexp> co

Syntax of designational expressions.

```
21.1    <id>  in  <label> co
21.2    <ui>  in  <label> co

21.3    <id>  in  <switch id> co

21.4    <bcs1>  im  <label1>          in  <decl label> is
                    <label1><bcs1>    in  <decl label> co

21.5    <bcs1>  im  <id1>          in  <decl switch id> is
                    <id1><bcs1>    in  <decl switch id> co

21.6    <label> b c in  <decl label> is o co

21.7    <id> b c  in  <decl switch id> is o co

21.8    <label1><bcs1><bc>  in  <decl label>  is
        <label1><bcs1>      in  <decl label>  co

21.9    <id1><bcs1><bc>  in  <decl switch id> is
        <id1><bcs1>      in  <decl switch id> co

21.10   formal <id1><bcs1> im <id1><bcs1> in <decl label> co
21.11   formal <id1><bcs1> im <id1><bcs1> in <decl switch id> co

21.12   label <label1><bcs1>  im  <label1><bcs1>  in  <decl label> co

21.13   switch <id1><bcs1>  im  <id1><bcs1>  in  <decl switch id> co

21.14   <label>        in  <sdexp> co
21.15   <switch des>   in  <sdexp> co
21.16   (<dexp>)       in  <sdexp> co

21.17   <decl label>        in  <decl sdexp> co
21.18   <decl switch des>   in  <decl sdexp> co
21.19   (<decl dexp>)       in  <decl sdexp> co

21.20   <sdexp>         in  <dexp> co
21.21   <decl sdexp>    in  <decl dexp> co

21.22   if <bexp> then <sdexp> else <dexp>  in  <dexp> co

21.23   if <decl bexp> then <decl sdexp> else <decl dexp>
        in <decl dexp> co

21.24   <switch id>[<sub exp>]  in  <switch des> co

21.25   <decl switch id>[<decl sub exp>]  in  <decl switch des> co
```

The value of boolean expressions and of arithmetic expressions

22.1 (<aexp1>) is <aexp1> co
22.2 (<bexp1>) is <bexp1> co

22.3 if <bexp1> then <saexp1> else <aexp1>
is
if va (<bexp1>) then <saexp1> else <aexp1> co

22.4 if <bexp1> then <sbexp1> else <bexp2>
is
if va (<bexp1>) then <sbexp1> else <bexp2> co

22.5 if true then <saexp1> else <aexp> is <saexp1> co
22.6 if true then <sbexp1> else <bexp> is <sbexp1> co

22.7 if false then <saexp> else <aexp1> is <aexp1> co
22.8 if false then <sbexp> else <bexp1> is <bexp1> co

22.9 <saexp1><rel op1><saexp2>
is
va (<saexp1>)<rel op1> va (<saexp2>) co

22.10 ¬ <bprimary1> is ¬ va (<bprimary1>) co

22.11 <bfactor1> ∧ <bsecondary1>
is
va (<bfactor1>) ∧ va (<bsecondary1>) co

22.12 <bterm1> ∨ <bfactor1>
is
va (<bterm1>) ∨ va (<bfactor1>) co

22.13 <implication1> ⊃ <bterm1>
is
va (<implication1>) ⊃ va (<bterm1>) co

22.14 <sbexp1> ≡ <implication1>
is
va (<sbexp1>) ≡ va (<implication1>) co

22.15 ¬ true is false co
22.16 ¬ false is true co

22.17 true ∧ true is true co
22.18 true ∧ false is false co
22.19 false ∧ true is false co
22.20 false ∧ false is false co

22.21 true ∨ true is true co
22.22 true ∨ false is true co
22.23 false ∨ true is true co
22.24 false ∨ false is false co

22.25 true ⌉ true is true co
22.26 true ⌉ false is false co
22.27 false ⌉ true is true co
22.28 false ⌉ false is true co

22.29 true = true is true co
22.30 true = false is false co
22.31 false = true is false co
22.32 false = false is true co

22.33 <int1> < <int2> is ¬ <int2> ≤ <int1> co
22.34 <int1> ≥ <int2> is <int2> ≤ <int1> co
22.35 <int1> > <int2> is ¬ <int1> ≤ <int2> co

22.36 <int1> = <int2> is <int1> ≤ <int2> ∧ <int2> ≤ <int1> co

22.37 <int1> ≠ <int2> is ¬ <int1> = <int2> co

22.38 <int1> ≤ <int2> is va (<int1> − <int2>) ≤ 0 co

22.39 − <ui> ≤ 0 is true co
22.40 <ui> ≤ 0 is false co
22.41 <ze> ≤ 0 is true co

22.42 + − <ui1> is − <ui1> co
22.43 − − <ui1> is <ui1> co

22.44 <int1> + − <ui1> is <int1> − <ui1> co
22.45 <int1> − − <ui1> is <int1> + <ui1> co
22.46 <int1> − + <ui1> is <int1> − <ui1> co

22.47 <factor1> ↑ <primary1>
is
<factor1> ↑ va (<primary1>) co

22.48 <term1><mult op1><factor1>
is
va (<term1>)<mult op1> va (<factor1>) co

22.49 <pm1><term1> is <pm1> va (<term1>) co

22.50 <saexp1><pm1><term1>
is
va (<saexp1>)<pm1> va (<term1>) co

22.51 <factor1> ↑ <ui1> is <factor1> ↑ (<ui1> – 1) × <factor1> co

22.52 <factor1> ↑ <ze>
is
if <factor1> ≠ 0 then 1 else <factor1> ↑ (– 1) co

22.53 <factor1> ↑ <ze> 1 is <factor1> co

22.54 <int1> : – <ui1> is – va (<int1> : <ui1>) co

22.55 <ui1> : <ui2>
is
if <ui1> < <ui2> then 0 else 1 + (<ui1> – <ui2>) : <ui2> co

22.56 <int1> × – <ui1> is – va (<int1> × <ui1>) co

22.57 <ui1> × <ui2> is <ui1> × (<ui2> – 1) + <ui1> co
22.58 <ui> × 0 is 0 co

22.59 <di><ui> in <ui> co

22.60 <pm><ui> in <int> co

22.61 0 <ze> in <ze> co

22.62 – <ui1> + <ui2> is <ui2> – <ui1> co

22.63 – <ui1> – <ui2> is – va (<ui1> + <ui2>) co

22.64 <ui1><di1><pm1><ui2><di2>
is
va (<ui1><pm1><ui2>) 0 + va (<di1><pm1><di2>) co

22.65 <ui1><di1><pm1><di2> is <ui1> 0 + va (<di1><pm1><di2>) co

22.66 <di1><pm1><ui1><di2> is <pm1><ui1> 0 + va (<di1><pm1><di2>) co

22.67 <ui1> 0 + <di1> is <ui1><di1> co
22.68 <di1> + <ui1> 0 is <ui1><di1> co

22.69 <ui1> 0 – <di1> is va (<ui1> – 1) 0 + va (10 – <di1>) co

22.70 10 – <di1> is 9 – va (<di1> – 1) co

22.71 <di1><pm1><di2> is va (<di1><pm1> 1) <pm1> va (<di2> – 1) co

22.72 <ui1><pm><ze> is <ui1> co

22.73 <ze><pm1><ui1> is <pm1><ui1> co

22.74 0 + 1 is 1 co
22.75 1 + 1 is 2 co
22.76 2 + 1 is 3 co
22.77 3 + 1 is 4 co
22.78 4 + 1 is 5 co
22.79 5 + 1 is 6 co
22.80 6 + 1 is 7 co
22.81 7 + 1 is 8 co
22.82 8 + 1 is 9 co
22.83 9 + 1 is 10 co

22.84 1 − 1 is 0 co
22.85 2 − 1 is 1 co
22.86 3 − 1 is 2 co
22.87 4 − 1 is 3 co
22.88 5 − 1 is 4 co
22.89 6 − 1 is 5 co
22.90 7 − 1 is 6 co
22.91 8 − 1 is 7 co
22.92 9 − 1 is 8 co

Basic symbols and auxiliary symbols.
Comment conventions.

23.1 <let> in <id> co
23.2 <id><let> in <id> co
23.3 <id><di> in <id> co

23.4 <int> in <constant> co
23.5 <logical value> in <constant> co

23.6 0 in <di> co
23.7 1 in <di> co
23.8 2 in <di> co
23.9 3 in <di> co
23.10 4 in <di> co
23.11 5 in <di> co
23.12 6 in <di> co
23.13 7 in <di> co
23.14 8 in <di> co
23.15 9 in <di> co

23.16 − <ui1> is ≮ − <ui1> ≯ co
23.17 + <ui1> is ≮ <ui1> ≯ co
23.18 <ui1> is ≮ <ui1> ≯ co

23.19 <pm><ze> is ≮ 0 ≯ co

23.20 true in <logical value> co
23.21 false in <logical value> co

23.22 <logical value1> is ≮ <logical value1> ≯ co

23.23 a in <let> co
23.24 b in <let> co
23.25 c in <let> co
23.26 d in <let> co
23.27 e in <let> co
23.28 f in <let> co
23.29 g in <let> co
23.30 h in <let> co
23.31 i in <let> co
23.32 j in <let> co
23.33 k in <let> co
23.34 l in <let> co
23.35 m in <let> co
23.36 n in <let> co
23.37 o in <let> co
23.38 p in <let> co
23.39 q in <let> co

23.40	r	in	<let>	co
23.41	s	in	<let>	co
23.42	t	in	<let>	co
23.43	u	in	<let>	co
23.44	v	in	<let>	co
23.45	w	in	<let>	co
23.46	x	in	<let>	co
23.47	y	in	<let>	co
23.48	z	in	<let>	co
23.49	A	in	<let>	co
23.50	B	in	<let>	co
23.51	C	in	<let>	co
23.52	D	in	<let>	co
23.53	E	in	<let>	co
23.54	F	in	<let>	co
23.55	G	in	<let>	co
23.56	H	in	<let>	co
23.57	I	in	<let>	co
23.58	J	in	<let>	co
23.59	K	in	<let>	co
23.60	L	in	<let>	co
23.61	M	in	<let>	co
23.62	N	in	<let>	co
23.63	O	in	<let>	co
23.64	P	in	<let>	co
23.65	Q	in	<let>	co
23.66	R	in	<let>	co
23.67	S	in	<let>	co
23.68	T	in	<let>	co
23.69	U	in	<let>	co
23.70	V	in	<let>	co
23.71	W	in	<let>	co
23.72	X	in	<let>	co
23.73	Y	in	<let>	co
23.74	Z	in	<let>	co
23.75	+	in	<spec del>	co
23.76	−	in	<spec del>	co
23.77	×	in	<spec del>	co
23.78	:	in	<spec del>	co
23.79	$\overline{\uparrow}$	in	<spec del>	co
23.80	<	in	<spec del>	co
23.81	>	in	<spec del>	co
23.82	≤	in	<spec del>	co
23.83	≥	in	<spec del>	co
23.84	=	in	<spec del>	co
23.85	≠	in	<spec del>	co
23.86	=	in	<spec del>	co
23.87	¬	in	<spec del>	co
23.88	∨	in	<spec del>	co

23.89 ∧ in <spec del> co
23.90 ¬ in <spec del> co
23.91 goto in <spec del> co
23.92 if in <spec del> co
23.93 then in <spec del> co
23.94 for in <spec del> co
23.95 do in <spec del> co
23.96 , in <spec del> co
23.97 : in <spec del> co
23.98 := in <spec del> co
23.99 step in <spec del> co
23.100 until in <spec del> co
23.101 while in <spec del> co
23.102 (in <spec del> co
23.103) in <spec del> co
23.104 [in <spec del> co
23.105] in <spec del> co
23.106 begin in <spec del> co
23.107 own in <spec del> co
23.108 array in <spec del> co
23.109 switch in <spec del> co
23.110 label in <spec del> co
23.111 value in <spec del> co

23.112 comment in <spec del> co
23.113 integer in <spec del> co
23.114 boolean in <spec del> co
23.115 procedure in <spec del> co

23.116 <let> in <end comment symbol> co
23.117 <di> in <end comment symbol> co
23.118 <logical value> in <end comment symbol> co
23.119 <spec del> in <end comment symbol> co

23.120 <end comment symbol><sequence of basic symbols
not containing semicolon or end or else>
in <sequence of basic symbols
not containing semicolon or end or else> co

23.121 end <sequence of basic symbols
not containing semicolon or end or else>
in <ext end> co

23.122 end in <comment symbol> co
23.123 else in <comment symbol> co
23.124 <end comment symbol> in <comment symbol> co

23.125 <comment symbol>
<sequence of basic symbols not containing semicolon>
in <sequence of basic symbols not containing semicolon> co

```
23.126 comment <sequence of basic symbols not containing semicolon>;
       in <comment> co

23.127 ;<comment> in <ext semicolon> co

23.128 begin <comment> in <ext begin> co

23.129 )<letter sequence> : ( in <ext par delimiter> co

23.130 <let><letter sequence> in <letter sequence> co

23.131 ;                     in <basic symbol> co
23.132 <comment symbol>      in <basic symbol> co

23.133 <basic symbol><sequence of basic symbols>
       in <sequence of basic symbols> co

23.134 <sequence of basic symbols1><ext par delimiter>
       <sequence of basic symbols2>
       is
       <sequence of basic symbols1> ,
       <sequence of basic symbols2> co

23.135 <sequence of basic symbols1><ext semicolon>
       <sequence of basic symbols2>
       is
       <sequence of basic symbols1> ;
       <sequence of basic symbols2> co

23.136 <sequence of basic symbols1><ext begin>
       <sequence of basic symbols2>
       is
       <sequence of basic symbols1> begin
       <sequence of basic symbols2> co

23.137 <sequence of basic symbols1><ext end>
       <sequence of basic symbols2>
       is
       <sequence of basic symbols1> end
       <sequence of basic symbols2> co

23.138 <basic symbol> in
       <basic symbol different from letter or digit> co

23.139 <let> in
       <basic symbol different from letter or digit> is false co

23.140 <di> in
       <basic symbol different from letter or digit> is false co
```

23.141 <sequence of basic symbols1>
<basic symbol different from letter or digit1><ze><ui1>
<sequence of basic symbols2>
is
<sequence of basic symbols1>
<basic symbol different from letter or digit1><ui1>
<sequence of basic symbols2> co

23.142 a in <aux term symb> co
23.143 b in <aux term symb> co
23.144 c in <aux term symb> co
23.145 d in <aux term symb> co
23.146 f in <aux term symb> co
23.147 g in <aux term symb> co
23.148 k in <aux term symb> co
23.149 l in <aux term symb> co
23.150 m in <aux term symb> co
23.151 o in <aux term symb> co
23.152 p in <aux term symb> co
23.153 t in <aux term symb> co

23.154 1 in <aux term symb> co
23.155 2 in <aux term symb> co
23.156 3 in <aux term symb> co
23.157 4 in <aux term symb> co

23.158 first in <aux term symb> co
23.159 of in <aux term symb> co
23.160 block in <aux term symb> co
23.161 formal in <aux term symb> co
23.162 actual in <aux term symb> co
23.163 save in <aux term symb> co
23.164 bn in <aux term symb> co
23.165 reset in <aux term symb> co
23.166 result in <aux term symb> co
23.167 bound in <aux term symb> co
23.168 pair in <aux term symb> co
23.169 within in <aux term symb> co
23.170 bounds in <aux term symb> co
23.171 not in <aux term symb> co
23.172 negative in <aux term symb> co
23.173 store in <aux term symb> co
23.174 eq in <aux term symb> co
23.175 subscript in <aux term symb> co
23.176 list in <aux term symb> co
23.177 go in <aux term symb> co
23.178 forbegin in <aux term symb> co
23.179 forend in <aux term symb> co
23.180 special in <aux term symb> co
23.181 equal in <aux term symb> co
23.182 length in <aux term symb> co

```
23.183  enter          in  <aux term symb> co
23.184  exit           in  <aux term symb> co
23.185  substitute     in  <aux term symb> co
23.186  function       in  <aux term symb> co
23.187  assign         in  <aux term symb> co
23.188  becomes        in  <aux term symb> co
23.189  zero           in  <aux term symb> co
23.190  sign           in  <aux term symb> co
23.191  dummy          in  <aux term symb> co
23.192  progr.p        in  <aux term symb> co
23.193  proc.body      in  <aux term symb> co

23.194  sign      in  <aux id> co
23.195  dummy     in  <aux id> co
23.196  dummy 1   in  <aux id> co
23.197  dummy 2   in  <aux id> co
23.198  forend    in  <aux id> co
23.199  <id> p    in  <aux id> co
23.200  <p> k     in  <aux id> co

23.201  <aux id>  in  <id> co

23.202  <p> k <ui>  in  <aux label> co
23.203  <p> l <ui>  in  <aux label> co
23.204  <p> m <ui>  in  <aux label> co

23.205  special label <p>  in  <aux label> co

23.206  <aux label>  in  <label> co

23.207  <basic symbol>
        in  <sequence of basic and aux term symbols> co

23.208  <aux term symb>
        in  <sequence of basic and aux term symbols> co

23.209  <sequence of basic and aux term symbols>
        <sequence of basic and aux term symbols>
        in
        <sequence of basic and aux term symbols> co

23.210  o  im  <sequence of basic and aux term symbols>  is  o co

23.211  o  im  <ass st>         in  <decl ass st>        is  o  co
23.212  o  im  <dexp>           in  <decl dexp>          is  o  co
23.213  o  im  <proc st>        in  <decl proc st>       is  o  co
23.214  o  im  <block>          in  <decl block>         is  o  co
23.215  o  im  <bplist>         in  <decl bplist>        is  o  co
23.216  o  im  <switch list>    in  <decl switch list>   is  o  co

23.217  o  is  ⊀ o ⊁
```

CHAPTER 6

EXPLANATION OF THE DEFINITION OF ALGOL 60

In this chapter we give an explanation of the techniques used in the metaprogram for the definition of ALGOL 60. Sections 1 to 6 contain some general comments, and sections 7, 8, ..., 30 correspond to sections 0, 1, ..., 23 of the metaprogram.

6.1. Defects of the definition

The following subjects have not been treated:

a. Real arithmetic.
   In ALGOL 60 no exact arithmetic has been specified ([38], 3.3.6); this specification belongs to the accompanying information which should be given by the programmer (cf. also [38], 1, footnote 1). Thus, whenever one wants to execute a program in which real arithmetic is used one has to extend the metaprogram with additional truths defining this arithmetic. Moreover, the declarator "real" should then be introduced and one should give the definition of its consequences for declarations, assignment statements, etc.
b. Procedure bodies in code and strings as actual parameters.
c. Standard functions (except the function "sign").

Another interpretation might be preferred in the following three cases:

a. Only the static definition of own is given (for the definition of this static interpretation see [1]).
b. Specifications of non value parameters are ignored.
c. The effect of a jump out of a function designator which leads to a label which is local to a function designator is defined in a way which differs from the usually accepted one. Details are given below.

In general, whenever in a program something occurs which was left undefined, said to be undefined or forbidden in [38], the value of the program is "ω"[1], which is used as a symbol for "undefined" (for more details see section 6.7). However, sometimes we could not avoid choice, e.g. regarding the order of evaluation of the value parameters, where we chose the order given in the formal parameter list. Also, primaries in the expressions are evaluated in order from left to right. A third example is the case of expressions containing formal parameters, which may become undefined if the corresponding procedure is called. Example: "<u>if</u> f <u>then</u> g <u>else</u> h", where "f", "g", and "h" are formal parameters, might be replaced by "<u>if</u> <u>true</u> <u>then</u> 3 <u>else</u> a ∨ b". Apparently, this does not fulfil the requirements of [38], 4.7.5. However, the metaprogram delivers "3" as the value of the last "expression".

The following two cases are treated incorrectly:

a. A conditional statement of the form "<u>if</u> <bexpl> <u>then</u> <unc stl>", where "<bexpl>" has the value "<u>false</u>", is equivalent to the dummy statement only if the evaluation of "<bexpl>" has no side effects.
b. Mutatis mutandis this holds for a goto statement leading to an undefined switch designator.

6.2. <u>Structure of the metaprogram</u>

The metaprogram of chapter 5 is used in the following way: Whenever one wants to evaluate an ALGOL 60 program, say "<program1>", the processor is asked to evaluate the following name:

⦃<LIST OF METAEXPRESSIONS CONTAINED IN CHAPTER FIVE>⦄ <u>co</u> <program1>.

The list of metaexpressions of chapter 5 is thus again directly added to V. As will be seen later, the evaluation of the simple name "<program1>" is subdivided into the evaluation of a list of simple names, roughly in such a way that each declaration or statement of the program corresponds to one simple name. Thus, during the evaluation of "<program1>", the

---

1) For easier readability, we use Greek letters in this chapter instead of the underlined Roman letters of chapter 5. Hence, "ω" corresponds to "<u>o</u>", "α" to "<u>a</u>", etc. (This convention is used only for single letters, not for underlined symbols containing more than one letter.)

metaprogram of chapter 5 is extended dynamically with new truths, each of which is the value of such a simple name. We distinguish two possibilities for the use of such a truth in a subsequent evaluation:

a. Direct application of the new truth.
   Example: The evaluation of the assignment statement "a := 3" will result in the addition to V of the truth

(1) a <u>is</u> 3

   (apart from some details concerning locality, which are given below; see also 4.2.3.3).
   Subsequent evaluation of the variable "a" will then lead to the application of the truth (1).

b. Indirect application of the new truth: The addition to V of a truth may have the effect that another truth becomes applicable to some simple name. Remember that the applicability of a truth containing a condition may depend on whether the derived condition of this truth envelopes another truth (cf. for example the Turing machine example, 3.1.2 and 4.2.2.2, where the applicability of the truths $T_{4,1}$ to $T_{4,5}$ depends on the truths corresponding to the quadruples in V_5). Many examples of this situation in the metaprogram for ALGOL 60 will follow.

The three main difficulties in the definition of ALGOL 60 proved to be:

a. The concept of locality.
b. The goto statements.
c. The requirement that all identifiers of a program be declared, even in parts of the program which are not executed. Thus, we have to consider e.g. "<u>begin</u> <u>if</u> <u>false</u> <u>then</u> i := 0 <u>end</u>" as an incorrect ALGOL 60 program.

The first point made it necessary to introduce the notion of block number, and the last two require the equivalent of a prescan.

In the evaluation of a program we distinguish the following phases:

0. Check on the syntactic correctness of the program. The metavariable "<program>" is defined in the metaprogram (T1.33 [1], T1.34, etc.) in such a way that it envelopes precisely the syntactically correct

---

[1] T, followed by a number, refers to the corresponding truth in chapter 5.

ALGOL 60 programs. In fact, that part of the metaprogram that defines the syntax of an ALGOL 60 program is essentially a transcription of the Backus notation in [38]. In establishing whether T2.1 is applicable to an ALGOL 60 program, the syntactic correctness of the program is thus checked automatically by the processor. The case that T2.1 is not applicable is considered in the section on undefined values; see section 6.7.

1. The prescan phase.

1.1. In the first phase of the prescan the different identifiers of the program, which are introduced either by explicit declaration, or by standing as a label or a formal parameter, are noted.

1.2. The second phase of the prescan checks whether each identifier in the program has been declared.

2. The execution phase.

2.1. In the first phase of the execution, the program is scanned for the occurrence of labels, which are then supplied with the block number of the smallest embracing block. This information makes it possible to restore the correct block number, if a goto statement leads out of a block.

2.2. Finally, the actual execution of the program takes place.

6.3. Determination of the block number

First we give an intuitive introduction to the definition and use of the block number.

Possible block numbers are (cf. T3.2 to T3.6):

"$\beta\gamma$", "$\beta\gamma\beta\gamma\beta\beta\gamma$", "$\beta\gamma\beta\beta\gamma$" or "$\beta\gamma\beta\gamma\beta\beta\gamma\delta\beta\gamma\beta\gamma$".

The following rules hold:

a. The γ's count block depth.

b. The β's between a certain γ and the immediately preceding γ count the number of parallel blocks at the depth of this γ.

c. The δ's count the depth of procedure calls.

At the beginning of the evaluation of a program, the block number is set to "$\beta\gamma$", i.e., "$\beta\gamma$" is added as a truth to V (first simple name of the right part of T2.1).

Next we consider the following example (we neglect for the moment the fact that a program is always embedded in a fixed outermost block, where some auxiliary declarations, e.g. of the function "sign", are made):

```
1: begin integer i; ...
         i := 0; ...
     2:L:begin integer j;
                ...
         end 2;
         i := i + 1; if i < 2 then goto L; ...
       3:begin integer k; ...
             4:begin integer 1;
                     ...
               end 4; ...
         end 3; ...
   end 1
```

In the prescan phase the block numbers are successively:

βγ	(initialized),
βγβγ	(by 1:begin),
βγβγβγ	(by 2:begin),
βγβγ	(by end 2),
βγβγββγ	(by 3:begin),
βγβγββγβγ	(by 4:begin),
βγβγββγ	(by end 4),
βγβγ	(by end 3),
βγ	(by end 1).

In the execution phase, the block numbers are successively (here we suppose that block 2 is executed twice):

βγ	(end prescan),
βγββγ	(by 1:begin),
βγββγβγ	(by 2:begin),
βγββγ	(by end 2),
βγββγββγ	(by 2:begin),

βγββγ	(by end 2),
βγββγβββγ	(by 3:begin),
βγββγβββγβγ	(by 4:begin),
βγββγβββγ	(by end 4),
βγββγ	(by end 3),
βγ	(by end 1).

The function of the δ's is the following:
If a procedure is called during the execution phase in a block with block number "<bn1>", and if this procedure is declared in a block with block number "<bn2>", then the block number is set to "<bn2> δ <bn1>". Upon exit from the procedure, "<bn1>" is activated again.

Example:

```
1:begin integer i;
       procedure P;
     2:begin integer j; ... end 2; P;
       ...
     3:begin integer k;
            ...; P; ...
       end 3; ...
  end 1
```

In the prescan the block numbers are successively:

βγ	(initialized),
βγβγ	(by 1:begin),
βγβγβγ	(by 2:begin),
βγβγ	(by end 2),
βγβγββγ	(by 3:begin),
βγβγ	(by end 3),
βγ	(by end 1).

In the execution phase the block numbers are successively:

βγ	(end prescan),
βγββγ	(by 1:begin),

βγββγβγ	(by 3:<u>begin</u>),
βγββγδβγββγβγ	(entrance to P),
βγββγββγδβγββγβγ	(by 2:<u>begin</u>),
βγββγδβγββγβγ	(by <u>end</u> 2),
βγββγβγ	(exit from P, cf. however, 6.24),
βγββγ	(by <u>end</u> 3),
βγ	(by <u>end</u> 1).

We now give a somewhat more precise description of the determination of the block number.

Each block entrance or exit, and each procedure entrance or exit in the execution phase, leads to addition to V of a new block number (a few other situations in which new block numbers are added to V will be treated below). At every moment, the last entry in V which has the syntactic form of a block number, is called the current (or active) block number. From the definition of applicability, it follows that it is always possible, by appropriate use of a condition in a truth, to find this last entry.

Suppose that, at a given moment, the current block number is "<bcs1><dbcs1>", and that a new block is entered. There are two possibilities: Either a truth of the form "<bcs1><bc1><dbcs>" occurs somewhere in V, meaning that the new block is parallel to an earlier one (possibly itself during the execution phase), in which case "<bcs1> β <bc1><dbcs1>" is added to V (T.6.4), or else no such truth is found, in which case "<bcs1> βγ <dbcs1>" is added to V (T6.3).

If the current block number is "<bcs1><bc><dbcs1>", then "<bcs1><dbcs1>" is added to V upon exit from the block (T6.2, T6.6). The value of the last "<u>end</u>" of the program, i.e., of the "<u>end</u>" of the fixed outermost block, is defined in T6.5 and explained below.

The rules governing block entrance and exit hold both for the prescan and execution phase of blocks and procedure bodies.

If a procedure is declared in a block with block number "<bn1>" and called from a block with block number "<bn2>", then "<bn1> δ <bn2>" is added to V (T17.77). After this the entrance to the procedure body (which is always made into a block) is performed according to the rules for block entrance given above.

Upon exit from a procedure, the current block number is looked up. This has the form"<bcs> δ <bnl>",and"<bnl>"is added to V (T17.78). By means of the last two rules, which of course only hold in the execution phase, the correct block number is available during execution of a procedure and after exit from this procedure.
In this way, at every moment the last truth in V which has the syntactic form of a block number defines the current block number. This is used whenever an identifier is processed; identifiers are always first extended with the current block number, so that e.g. uniqueness of identifiers is guaranteed in recursive situations.

Finally, we introduce the following terminology: The "significant part" of a block number is that part of the block number that precedes the left most"δ".If no"δ"is present, its significant part is itself. Usually, we are interested only in the significant part of the block number. Therefore, we often write "block number" where we should write "significant part of block number".

6.4. The prescan

As explained above, we have introduced two phases in the evaluation of an ALGOL 60 program, the prescan phase and the execution phase, each of which is subdivided again into two phases. Each block of the program passes once through the prescan phase, whereas the number of times it is executed is clearly determined dynamically. This structure of an evaluation of the program in several phases is not available directly in the metalanguage. However, the basic idea was already demonstrated in the example of 4.2.1.4. If a certain sequence of symbols is evaluated by means of a metaprogram, it is possible to introduce as a "side effect" of the evaluation of that sequence the addition to V of new truths in such a way that when precisely the same sequence is evaluated again, its value is different from the result of the first evaluation. This idea is used extensively in the prescan rules (T4.1 to T5.15), in view of the two problems mentioned above: the processing of goto statements and the check whether all identifiers of a program are declared.

The structure of the prescan is based on the concept of "program point" (defined syntactically as the metavariable "<p>" in T3.7). Essentially, the evaluation of the ALGOL 60 program is replaced by the evaluation of a sequence of program points in such a way that:

a. Each declaration or statement corresponds to precisely one program point. The uniqueness of the program point is achieved by defining it in such a way that its first part ("<bcs>") is equal to the block number of the block which is scanned, while its second part ("<as>") is different for each declaration or statement in this block (the declarations and statements are numbered successively in the order in which they occur in the program; see also 4.2.3.4).
b. The evaluation of a certain program point is defined differently for the several phases.

Next we give a more detailed explanation of T4.1, the main prescan rule for declarations.
First of all, we remark that the definition of "<block tail>" is given in T1.27. Note moreover that an example of a specific case of the left part of T4.1 is provided by each specific case of the right part of T2.1.
The right part of T4.1 consists of three simple names:

1. "<declaration1><p1> 1".
   This means that "<declaration1>", which occurs at program point "<p1>", has to be evaluated according to the rules which are given for the evaluation of a declaration in phase 1. E.g., if "<declaration1>" is a type declaration, T7.7 or T7.8 will prove to be applicable. The details of these rules will be explained below. However, we may already mention one essential point: The effect of the evaluation of a declaration in prescan phase 1 is that the identifier which is declared is added to V, supplied with the current block number and its type, so that it is known in phase 2 of the prescan that this identifier has been declared.
2. The evaluation of the second simple name of the right part of T4.1 results in the addition to V of:

```
      <p1> is {<declaration1><p1> 2 co
         ⦃<p1> is {
(1)         ⦃ τ <p1> is {<declaration1><p1> 4 co
              τ <p1> α}⦄ co
          <p1> α}⦄ co
      <p1> α}
```

Suppose now that phase 2 of the prescan is reached (how the transition to phase 2 is achieved is explained later). In this phase, as in phases 3 and 4 (i.e. the two phases of the execution), the sequencing of the evaluation of the different declarations and statements of the program is replaced by the evaluation of the successive corresponding program points,which is made possible by the addition of new truths, such as (1).

Suppose moreover, that"<p1>"is evaluated. Application of (1) then leads to the evaluation of:

2.1. "<declaration1><p1> 2". This means again that "<declaration1>", occurring at program point "<p1>", has to be evaluated, but now according to the rules for the evaluation of a declaration in phase 2 (see e.g. T7.9, T9.14 or T11.8). Since, as a result of phase 1, it is known which identifiers have been declared, it is now possible to check whether "<declaration1>" contains only declared identifiers.

2.2. The evaluation of the second simple name in (1) results in the addition to V of:

```
      <p1> is {
(2)      ⦃ τ <p1> is {<declaration1><p1> 4 co
           τ <p1> α}⦄ co
      <p1> α}
```

If phase 3 is reached, and supposing "<p1>" is evaluated again, application of (2) will result in:

2.2.1. Addition to V of

```
(3) τ <p1> is {<declaration1><p1> 4 co τ <p1> α}
```

(Note that the evaluation of "<declaration1><pl> 3" is missing. This is indeed unnecessary, since phase 3 is only introduced for the processing of labels.)

If phase 4 is reached, and supposing "τ <pl>" is evaluated (the reason for the extension of "<pl>" with the extra symbol "τ" will be given below), application of (3) will result in:

2.2.1.1. Evaluation of "<declaration1><pl> 4".
"<declaration1>", occurring at program point "<pl>", will now be evaluated according to the rules of phase 4 (e.g. T7.10, T9.15, etc.).

2.2.1.2. The evaluation will be continued by the evaluation of the next program point, i.e. of "τ <pl> α". Here we note the basic sequencing idea: The evaluation of a program point is always defined in such a way that its successor is evaluated as the next step; "τ <pl> α" is the program point corresponding to the declaration or statement which follows "<declaration1>" in the program.

2.2.2. Evaluation of "<pl> α"; remember that this results from application of truth (2).

2.3. Evaluation of "<pl> α"; this results from application of truth (1).

3. Evaluation of "<pl> α : <block tail1>"; this results from evaluating the final simple name of T4.1. If "<block tail1>" begins with a declaration, then T4.1 will be applied again. This will result in the same structure of additions to V, this time however with "<pl> α" instead of "<pl>".

From the example T4.1 the outline of the structure of the sequencing of the evaluations should have become clear:

a. By addition of new truths, program points are evaluated by application of different truths in different phases.
b. Sequencing is achieved by organizing the added truths in such a way that evaluation of a program point leads automatically to the evaluation of the next program point.

Of course, there remains the explanation of the way in which the transition between the different phases is accomplished.

As a second example we consider T5.7. This truth has almost the same structure as T4.1. However, we note four differences:

1. The metavariable "<block end>" is used instead of "<block tail>". The definition of "<block end>" is given in T1.32 (cf. also T1.28 to T1.31). It is essentially the same as the "<compound tail>" of [38], 4.1. Some complications were caused by the for statement (see section 6.23).
2. "<unlabelled basic st1><pl> <u>1</u>" is not included. In fact, in T4.1 the evaluation of "<declaration1><pl> <u>1</u>" in phase 1 leads to the addition to V of information about the declaration of the corresponding identifier(s). There is apparently no point in doing this here.
3. The successor of "τ <pl>" is missing:
   The innermost metastring has the form:
   "⦃ τ <pl> <u>is</u> <unlabelled basic st1>⦄" and not:
   "⦃ τ <pl> <u>is</u> {<unlabelled basic st1> <u>co</u> τ <pl> α}⦄".
4. An extra auxiliary label "<pl> κ" labels "<block end1>". See also section 6.6; the label "<pl> κ" should not be confused with the program point "<pl> α".

In order to explain the reasons for the differences mentioned in points 3 and 4, we consider the statement sequencing in somewhat more detail: We distinguish the following cases:

1. Three kinds of unlabelled basic statements, i.e., assignment statements, goto statements and procedure statements.
2. Blocks.
3. Conditional statements and compound statements.
4. For statements.
5. Dummy statements (note that by T1.1 to T1.3, a dummy statement is not an unlabelled basic statement).
6. Labelled statements.

Remarks:

1. In cases 2 to 5 above, we consider only the unlabelled statements.
2. The dummy statement is treated by:
   a. Appropriate use of optional metavariables.
   b. T5.1.

We shall not explain this in more detail.

3. The (complicated) treatment of the for statement is described in T16.1 to T16.32 and explained separately.
4. By applying T5.2 to T5.6, compound statements and conditional statements are replaced by sequences of goto statements and (possibly labelled) unconditional statements or for statements.

We now return to T5.7.

If "<unlabelled basic stl>" is an assignment statement, say "<ass stl>", the effect of applying T14.18 to "<ass stl><pl> 4" will be:

a. "<ass stl>" is evaluated.

b. "τ <pl> α" is evaluated.

Thus, we use here the same technique as with declarations: the successor of the assignment statement concerned is executed as a result of the evaluation of the corresponding program point.

It is possible that a jump out of a function designator, which occurs in the assignment statement, is executed as a result of step a. If no special measures were taken (described below), subsequent evaluation of "τ <pl> α" would then completely upset the correct order of statement execution. (A similar difficulty may arise e.g. in the evaluation of a bound pair list in an array declaration.)

Next, we suppose that "<unlabelled basic stl>" is a goto statement, e.g. "goto <labell>". Essentially, the result of the evaluation of "goto <labell>" is that the program point, corresponding to the statement which is labelled by "<labell>", is evaluated. In fact, one of the two main reasons for the introduction of the program points was the desire to make this solution of the processing of goto statements possible (cf. also 4.2.3.4, example 13). Details about the case in which the goto statement is not simply of the form "goto <labell>", and about the treatment of the (unfortunate) concept of the undefined dummy switch designator ([38], 4.3.5) are given below.

Finally, suppose that "<unlabelled basic stl>" is a procedure statement. Here we use the idea of A. van Wijngaarden, described in [49], which is equivalent to the following scheme (which is applied only to non type procedures):

a. The procedure declaration is supplied with an extra formal parameter.
b. A goto statement leading to this extra formal parameter is introduced at the end of the procedure body.
c. As corresponding actual parameter a label which labels the statement following the procedure statement is supplied. This is the reason for the introduction of the label "<pl> κ" in T5.7. (In the case that "<unlabelled basic stl>" is not a procedure statement, "<pl> κ" has no function.)

Since this process is not applied to type procedures a difficulty arises when a function designator is used as a statement. A solution for this case is described later.

Next we make some remarks on T5.8. Here we find the same structure as in T4.1 and T5.7. However, it appears that only in phases 1 and 3 need anything be evaluated: Phase 1 is used again to establish that "<labell>" is a "declared" identifier (or integer; the somewhat unusual notion of a declared integer is apparently introduced in the following sentence of [38], 4.1.3: a label separated by a colon from a statement, ..., behaves as though declared in the head of the smallest embracing block...). This information is then used in phase 2 in the check whether labels occurring indesignational expressions have been declared. In phase 3 "label <labell><pl> 3" is evaluated, as defined in T12.3 and T12.4. The effect is that a truth is added to V establishing a correspondence between "<labell>", "<pl>" and the current block number (for the meaning of "<fgsl>" see sections 6.22 and 6.23). As explained above, the correspondence between"<labell>" and "<pl>" is used in the evaluation of a goto statement, leading to "<labell>".

There remains the treatment of blocks in the prescan (T5.9 to T5.15, cf. also T2.1):

1. Phase 1 of the prescan of the program is initiated by the evaluation of the third simple name in the right part of T2.1.
2. Before the "end" of each block (except for the program itself and procedure bodies, but including blocks within procedure bodies), an extra goto statement is included, leading to an extra label, which labels the statement immediately following the block concerned.

This is achieved by application of T5.9. Infinite addition of extra goto statements is avoided by the introduction of the auxiliary metavariable "<special block>": Once the extra goto statement is added to the block, it becomes a special block, and T5.11 becomes applicable.

3. Application of T5.11 to a special block has the following effect:

3.1. No special actions are performed in phases 1 and 3.

3.2. In phase 2 the simple names "<special block1> in <decl block>" and "first progr.p of block <pl>" are evaluated. After this, the standard addition to V of a new truth for "<pl>" is performed, and the succeeding "<pl> α" is evaluated.
The evaluation of "<special block1> in <decl block>" leads to the prescan of "<special block1>" (T5.13, "<special block1>" is a specific case of "begin <block tail>"), i.e., the prescan mechanism is activated recursively for this block by T5.13; see also below.
The prescan of procedure bodies is performed by evaluating the appropriate simple names in the right parts of T13.14 to T13.17, as explained later.
By applying T5.12, the evaluation of "first progr.p of block <pl>" leads to addition to V of a truth which defines the first program point of the block corresponding to "<pl>".
One should note the difference in T5.12 between "<bcsl><asl>", which is the program point corresponding to the special block we consider, and "<bcsl><bcl> α", which is the program point corresponding to the first declaration in the block concerned. This declaration occurs one block level deeper, and therefore an extra "γ" (and one or more "β"'s) are needed; hence, the transition between "<bcsl><asl>" and "<bcsl><bcl> α".
Note moreover that "<bcsl><bcl>" was left in V as a result of the block entrance in the evaluation of "<special block1> in <decl block>", as will become clear later. However, at the moment of application of T5.12, "<bcsl><bcl>" is not the current block number, since this has been reset to "<bcsl>" upon exit from "<special block1>".

3.3. In phase 4 the evaluation takes place of "first progr.p. of block <pl>". The necessary preparations for this evaluation were made in phase 2, since:

a. As a result of the evaluation of "<special block1> in <decl block>" (i.e. of the prescan of this block), truths have been added to V which define the values of the successive program points corresponding to the declarations and statements of this block.

b. As a result of the evaluation of "first progr.p. of block <pl>" (in phase 2, described above), the first program point of the block concerned can be found in V.

This means that phase 3 of the evaluation of the block, corresponding to "<pl>", is started by the evaluation of "first progr.p. of block <pl>", in phase 4 of the evaluation of the smallest embracing block of the block concerned.

Note that no successor "τ <pl> α" is evaluated in phase 4. This is not necessary, since the extra goto statement which was added in T5.9, ensures the execution of the successor of the block concerned.

Finally, we describe the evaluation of "<special block1> in <decl block>", i.e., the way in which the prescan mechanism for a block is called recursively.

First of all, T5.13 will be applied; hence, the two simple names in its right part are evaluated. The first one, i.e. "begin", leads to the addition to V of the block number for this new block (as described in 6.3). The second simple name is evaluated by applying T5.14. The right part of T5.14 consists of two simple names.

Evaluation of the first simple name leads to addition to V of:

<bcs1> α is {
    ⟨<bcs1> α is {begin co
        ⟨ τ <bcs1> α is τ <bcs1> αα⟩ co
    <bcs1> αα}⟩ co
<bcs1> αα

This is a truth of the same structure as those resulting from T4.1, T5.7, etc. We see that no special actions are taken in phases 1, 2 or 4, but only in phase 3, where "begin" is evaluated again: This time the dynamic block introduction during execution time is performed.
The second simple name of the right part of T5.14, i.e., "<bcs1> αα : <block tail1>", has precisely the form to which one of T4.1, T5.7, etc. is applicable. Hence, evaluation of this simple name starts the prescan of the block concerned.

Repeated application of the prescan rules T4.1, T5.7, etc. to a given block, will eventually lead to exhaustion of the sequence of declarations and statements in this block, after which a program point, corresponding to its "end", is introduced. Then T5.15, which is important for the transition between the several phases, will be applicable.
We describe its right part in detail:
1. Evaluation of its first simple name leads to addition to V of:

    <bcs1><as1> is {end co
        ⦃<bcs1><as1> is {
            ⦃ τ <bcs1><as1> is end⦄ co
            τ <bcs1> α}⦄}

1.1. Evaluation of "<bcs1><as1>" in phase 2 has the following effect:
1.1.1. "end" is evaluated. This leads to the block exit; phase 2 of the prescan of the block concerned is now finished.
1.1.2. To V is added:

(1) <bcs1><as1> is {⦃ τ <bcs1><as1> is end⦄ co τ <bcs1> α}

Note that there is no successor of "<bcs1><as1>" in phase 2. Evaluation of "<bcs1><as1>" in phase 3 leads to application of (1), with the following effect:
1.1.2.1. To V is added

(2) τ <bcs1><as1> is end

1.1.2.2. "τ <bcs1> α" is evaluated.
The evaluation of "τ <bcs1> α" starts phase 4 of the block concerned; here we find the transition from phase 3 to phase 4.

The reason for the extra symbol "τ" can now be given: If this symbol had not been introduced, it would have been impossible to distinguish between phase 3 and phase 4, if the block concerned is executed more than once.
Evaluation of "τ <bcsl><asl>" in phase 4 leads to application of (2), hence "end" is evaluated, which means that the (dynamic) block exit is performed.
Again, there is no successor given of "τ <bcsl><asl>" in phase 4. If the block concerned is not the body of a type procedure, its successor is found by the extra goto statement, added in T5.9 (thus, in this case "τ <bcsl><asl>" will in fact never be evaluated), whereas in the case that the block is the body of a type procedure, there is of course no succeeding statement.

2. Evaluation of the second simple name of the right part of T5.15, i.e., of "<bcsl> α", starts phase 2 of the prescan; here we note the transition from phase 1 to phase 2.

We now summarize the rules about initiation of and transition between the different phases:

1. Phase 1 of the prescan of the program is initiated by the evaluation of the third simple name of the right part of T2.1.
2. Phase 1 of the prescan of all other blocks is initiated by evaluating "<special blockl> in <decl block>" (T5.11; the similar case of procedure bodies is treated by T13.14 to T13.17).
3. Transition from phase 1 to phase 2 is performed by application of T5.15.
4. Phase 3 of the program is initiated by the evaluation of the first program point of the program, i.e., of "βγα" (fourth simple name of the right part of T2.1).
5. Phase 3 of the execution of inner "normal" blocks (i.e. blocks other than procedure bodies) is initiated by the evaluation of "first progr.p. of block <pl>", in phase 4 of the evaluation of the smallest embracing block (T5.11).
6. Phase 3 of the execution of procedure bodies is initiated by a mechanism explained later.

7. The transition from phase 3 to phase 4 is performed by application of T5.15.

6.5. The requirement that all identifiers of a program be declared

In phase 2, application of T4.1 and T5.7 leads to evaluation of "<declaration><pl> 2" and "<unlabelled basic stl><pl> 2" (evaluation, in phase 2, of "<special blockl> in <decl block>" results in the same simple names).

Depending upon the different kinds of declarations and unlabelled basic statements, the following possibilities arise:

a. <type declarationl><pl> 2
b. <array declarationl><pl> 2
c. <switch declarationl><pl> 2
d. <procedure declarationl><pl> 2
e. <ass stl><pl> 2
f. goto <dexpl><pl> 2
g. <proc stl><pl> 2

Clearly, it is not necessary to check whether identifiers occurring in type declarations have been declared. Hence, the definition of T7.9, which simply leads to the addition of "tr" to V.

(This is a device which is used often in the metaprogram; addition of "tr" to V has no influence on the rest of the evaluation of the program. The reason for inclusion of T7.9 is the desire to obtain a uniform treatment of declarations in T4.1; various reasons for addition of "tr" to V in several other cases will appear in the sequel.)

By T9.14, "<array declarationl><pl> 2" also has the value "tr". The check whether the identifiers occurring in the bound pair lists have been declared is already performed in phase 1, since these identifiers have to be declared in embracing blocks and not in the block itself in which the array declaration occurs. Details of this check are given later.

By T11.8, evaluation of a switch declaration in phase 2 leads to evaluation of "<switch listl> in <decl switch list>", where "<switch listl>" is the switch list occurring in the switch declaration concerned.

The more complicated treatment of procedure declarations in phase 2 is explained below.

By T14.17, T15.1 and T17.62, evaluation of "<ass stl><pl> 2", "goto <dexpl><pl> 2", and "<proc stl><pl> 2" leads to evaluation of "<ass stl> in <decl ass st>", "<dexpl> in <decl dexp>", and "<proc stl> in <decl proc st>" respectively.

Next we explain the way in which the simple names "<switch list> in <decl switch list>", ..., "<proc st> in<decl proc st>" are evaluated.

The main features of the evaluation of these simple names are:

a. Inclusion in the metaprogram, in addition to the truths which are equivalent to the BNF rules of [38], of related truths, such as:
   "<decl factor> in <decl term>" besides "<factor> in <term>",
   "<decl aexp> in <decl sub exp>" besides "<aexp> in <sub exp>",
   "<decl int var> in <decl primary>" besides "<int var> in <primary>",
   "<decl saexp><rel op><decl saexp> in <decl bprimary>" besides
   "<saexp><rel op><saexp> in <bprimary>",
   etc.

b. Use of a search in embracing blocks, by means of the block number.

c. Use of information which is added to V in phase 1.

As a result of these three points, the process of checking whether all identifiers of a program are declared is performed automatically by the processor, as follows from the definition of envelope and applicability.

We give an example:

In order to evaluate "a := b + c in <decl ass st>", T14.15 is eventually tried for applicability (the preceding truths, in particular T14.16 will prove to be inapplicable). T14.15 is applicable, if "a in <decl int left part list>" and "b + c in <decl aexp>" have the value "tr".

Application of T14.11, T14.3 and T18.33, to "a in <decl int left part list>" leads to evaluation of "a in <decl int var id>".

Application of T19.28, T19.26, T19.24, T19.19, T19.15, T19.10 and T18.33 to "b + c in <decl aexp>" leads to evaluation of "b in <decl int var id>" and "c in <decl int var id>". Application of T18.11 to "a in<decl int var id>" leads to evaluation of "a <bcsl> in<decl int var id>", where "<bcsl>" is the current block number.

If "a" has been declared in this same block, application of T18.27 results in the value"tr"for "a in <decl int var id>". (As will be seen later, if "a" is an integer variable, declared in the block with block number "<bcs1>", there will have been left in V a truth of the form "integer a <bcs1>", as a result of phase 1. Since the condition of T18.27 envelopes this truth, T18.27 is applicable to "a <bcs1> in <decl int var id>".)
If "a" is not declared in the block with block number "<bcs1>", it might be a formal parameter, in which case T18.23 applies (the way in which "formal a <bcs1>" might have been added to V is again described later).
If "a" is not a formal parameter either, then by T18.19 the evaluation of "a <bcs2><bc1> in <decl int var id>" (with"<bcs2><bc1>"="<bcs1>"),is replaced by the evaluation of "a <bcs2> in <decl int var id>", i.e., the smallest embracing block is now considered, and T18.27 and T18.23 are tried again (note that the current block number is not changed if an embracing block is tried).
In case of no success, by repeated application of T18.19, all embracing blocks are searched, until there is no longer an embracing block. Then T18.15 applies, and the value of "a in <decl int var id>" is some symbol, viz."ω",different from"tr",which means ultimately that T14.15 is not applicable to "a := b + c in <decl ass st>".
If, on the other hand, "a" and also "b" and "c", have been declared correctly, the final result is that "a := b + c in <decl ass st>" has the value"tr",which is then the result of the evaluation of "a := b + c <pl> 2" in phase 2. Again, addition of"tr"to V does not influence the rest of the evaluation.
The case that one of the identifiers in "a := b + c" turns out not to be an integer variable nor a formal parameter, will be treated below (6.7). In a sense, the evaluation of the program is then stopped.

From the given example, it follows that phase 2 is in fact not only used for the test whether all identifiers have been declared, but also to check that the identifiers have the correct types. With formal parameters, this check is of course in general impossible. Therefore, the type of a formal parameter is always considered to be correct.

All evaluations in phase 2 proceed essentially as in the above given example.

6.6. Auxiliary identifiers and labels

In several places we have introduced auxiliary identifiers and labels, such as: the identifiers "dummy 1", "dummy 2" and "sign" in T2.1, the labels "<p> λ 1" to "<p> λ 4" in T5.3 to T5.6, the label "<p> κ" in T5.7, etc. (a complete list is given in T23.194 to T23.200 and T23.202 to T23.205). By T23.201 and T23.206, these auxiliary identifiers and labels are indeed identifiers and labels; hence, the metaprogram will treat simple names which contain these auxiliary identifiers and labels in the same way as simple names containing "normal" (i.e. defined as in [38]) identifiers and labels. However, there is one exception to this rule: If an auxiliary identifier or label occurs in the original program, then by T2.2 or T2.3, its value is defined to be "ω" (6.7).

6.7. "Undefined values" (T0.1 to T0.7)

Whenever in the course of the evaluation of a program something occurs which was left undefined, said to be undefined or forbidden in [38], we have tried to arrange that the value of the program is then "ω". It is, however, in general impossible to deliver the single symbol "ω" as the value of the whole program (with two exceptions, see below), since, as a result of the prescan, the evaluation of the program is divided into the evaluation of a list of simple names. Thus, the value of the program is necessarily the list of the values of these simple names. The best we could do was to try to organize the evaluation of the program in such a way that essentially, whenever the value of a certain simple name happens to be "ω", that then the values of the remaining simple names are also "ω", so that the value of the program terminates with a list of "ω"'s.

We now give more details about our treatment of the "undefined values". First we treat the case that the "program" which is evaluated contains an auxiliary identifier or label. Then, by T2.2 or T2.3, its value is defined to be "ω" (provided it consists only of basic symbols or auxiliary terminal symbols (see below)).

If the program does not contain an auxiliary identifier or label, but it is syntactically incorrect for some other reason, then T0.1 will be

applied, again with the result "ω".
(Note that the introduction of an auxiliary identifier or label results in a program which is syntactically incorrect in the sense of [38], but which is still a specific case of "<program>" as defined in the metaprogram. Hence, the need for truths T2.2 and T2.3.)
The syntactic definition of a sequence of basic and aux(iliary) term(inal) symbols is given in T23.207 to T23.209. The auxiliary terminal symbols are listed in T23.142 to T23.193. Examples of their use have already been given in the definition of a block number, a program point, the symbols "1", "2", "3", "4", which indicate evaluations in the different phases, the simple name "first progr.p. of block <p>", etc.
As explained above, the evaluation of a syntactically correct ALGOL 60 program is divided into the evaluation of a list of simple names. Each of these simple names (except for the metastrings) is either a sequence of basic and auxiliary terminal symbols, or of one of the forms "<ass stl> in <decl ass st>", ..., "<switch listl> in <decl switch list>".
(Remember that simple names of the second kind where introduced in phase 2.)
The metaprogram is organized in such a way that whenever one of these simple names contains something which was left undefined, said to be undefined or forbidden in [38] (e.g. an undeclared identifier, an identifier which is declared more than once in the same block, an array element with subscripts outside the array bounds, number of actual parameters in a procedure call different from the number of formal parameters, etc.), then the value of that simple name is either directly defined to be "ω" (see e.g. T7.8), or none of the truths except one of T0.1 to T0.7 will be applicable.
Suppose now that the evaluation of a certain simple name has resulted in the addition of "ω" to V; from then on, all remaining simple names (for an exception see below) will also have the value "ω", since one of T23.210 to T23.216 will now be applicable: The addition of "ω" to V has the effect that the condition in T23.210 to T23.216 has the value "tr". Thus, once a certain simple name has the value "ω", all other simple names will have the value "ω".

There is, however, an exception to this rule: As follows from the definition of the metalanguage, metastrings are not evaluated by applying the metaprogram; hence, the occurrence of "ω" in V will not prevent the addition of the values of these metastrings to V.

6.8. Syntax of a program (T1.1 to T1.34)

The truths in this section are essentially equivalent to the BNF rules for an ALGOL 60 program, cf. [38], 4.1.1, etc. Some minor modifications were needed for the treatment of the dummy statement.
Also, the metavariables "<block end>" and "<block tail>" were introduced for subsequent use, e.g. in the prescan rules. The reason for the unusual definition of "<block end>" will become clear when we treat the for statement.

6.9. Value of a program (T2.1 to T2.3)

The main aspects of T2.1 have already been treated in the description of the prescan.
The first simple name of its right part initializes the block number.
The second simple name initializes a for counter, which is used in the definition of the for statement and is explained later.
The third simple name initiates the prescan. Note that the given program is embedded into an outermost block which contains auxiliary declarations.
The reason for the type declarations "integer dummy 1" and "boolean dummy 2" will become clear below. The integer procedure "sign" is introduced in view of the definition of the for statement.
Evaluation of the fourth simple name starts the execution of the program.
T2.2 and T2.3 were treated above.

6.10. Syntax of block number and program point (T3.1 to T3.7)

This requires no special comment.

6.11. Prescan declarations (T4.1 to T4.3)

T4.1 has been explained already.

By means of T4.2, a type declaration, containing more than one identifier, is replaced by a sequence of type declarations, each containing only one identifier.
T4.3 has a similar function for array declarations.

6.12. Prescan statements (T5.1 to T5.15)

The meaning of T5.1 and T5.2 is clear.
T5.3 to T5.6 are used to transform a conditional statement into a sequence of unconditional statements or for statements.
T5.7 to T5.15 have been treated already in the description of the prescan mechanism.

6.13. Value of begin and end (T6.1 to T6.6)

Except for T6.5, these truths were treated above.
T6.5 needs some more explanation. We have already mentioned that jumps out of function designators occurring in expressions can upset the correct order of evaluation of a program: For example, let "<pl>" correspond to an assignment statement; then from T14.18 it follows that after the completion of the evaluation of this assignment statement, "τ <pl> α" has to be evaluated. However, if a jump out of this assignment statement occurs, we have to find a way to avoid subsequent evaluation of "τ <pl> α". This is accomplished by the following device:

a. Each block ends with a goto statement, leading to the successor of this block; hence, a jump out of a function designator leads to the evaluation of the whole rest of the program (for an exception see below), and only after completion of the whole program will the evaluation of T14.18 be continued by evaluating "τ <pl> α".
b. However, application of T6.5 will result in addition to V, in phase 4, of the truth
"<sequence of basic and aux term symbols>".
Thus, after "completion" of the program, every simple name which is evaluated afterwards, such as "τ <pl> α", has the value "tr". This means that we have in a way cancelled the superfluous evaluations after the actual completion of the program.

The above described scheme does not work for a jump out of a function designator to a label which is local to a function designator, cf. 6.1.

6.14. Type declarations (T7.1 to T7.13)

The meaning of T7.1 to T7.6 is obvious.

T7.7 leads to addition to V of the identifier concerned, supplied with its type and the current block number. However, if the same identifier has been declared already in this block, then T7.8 will be applicable and "ω" is added to V. In fact, if "<idl>" has been declared already in this block, then a truth will have been added to V which is enveloped by "<specifier><idl><bcsl>", whence the applicability of T7.8. For the definition of "<specifier>" see T13.4 to T13.7.

By T7.10 and T7.11, evaluation of a declaration of a non own simple variable in phase 4 leads again to addition to V of the identifier concerned, supplied with its type and the current block number (the block number in phase 4 is of course different from that in phase 2).

T7.12 and T7.13 treat the somewhat more complicated case of own type declarations, e.g. "<u>own</u> <typel><idl><pl> <u>4</u>". Two cases are distinguished:

1. If the block in which the declaration of the own simple variable occurs is executed for the first time, T7.12 will apply; hence, two truths are added to V:

(1) <typel><idl><bcsl>

This is just the same as with a non own simple variable.

(2) τ <pl> <u>is</u> {<u>own</u> <typel><idl><pl> <u>4</u> <bcsl> <u>co</u> τ <pl> α}

(2) has the following effect:

2. If the block in which the declaration of the own simple variable occurs is executed again, then the program point corresponding to this declaration will be evaluated by applying truth (2):

2.1. Evaluation of the first simple name of the right part of (2) is performed by applying T7.13. Again, two simple names are evaluated:

2.1.1. The first simple name of the right part of T7.13 is of the same form as an own declaration which is executed for the first time (see 1. above).

2.1.2. The second simple name leads to addition to V of

(3) <idl><bcsl> is <idl><bcs2>

Here "<bcsl>" is the block number of the current activation of the block concerned, "<bcs2>" is the block number of its previous activation. The effect of (3) is that if "<idl>" is evaluated at the moment that "<bcsl>" is the current block number, and if there has been no assignment to "<idl>" during this activation of the block, then evaluation of "<idl><bcsl>" is replaced by evaluation of "<idl><bcs2>"; i.e., the processor now searches for a value of"<idl>" which was possibly assigned to it in the previous activation of the block concerned, which had as its block number "<bcs2>". (In order to understand this mechanism completely, one also has to know how the evaluation of a simple variable and of an assignment statement is defined.)

2.2. Evaluation of the second simple name of the right part of (2) will, as usual, lead to evaluation of the declaration or statement which follows the own type declaration.

6.15. The value of a simple variable (T8.1 to T8.6)

If a simple variable, say "<idl>", is evaluated, and if the current block number is "<bcsl>", then application of T8.1 results in evaluation of "<idl><bcsl>".

If "<idl>" has been declared in the block with block number "<bcsl>", and if T8.4 proves to be applicable, then no assignment to "<idl>" has taken place in this block, for otherwise first a dynamically added truth of the form "<idl><bcsl> is <intl>" or "<idl><bcsl> is <logical valuel>" would have been met as the result of such an assignment (see also the definition of assignment statements). Thus, applicability of T8.4 indicates that the simple variable concerned did not get a value in this block, whence its value is defined to be "ω".

Another possibility is that "<idl><bcsl>" is a formal parameter, called by name, which has an expression "<expl>" as its corresponding actual parameter. Then T8.3 will be applicable. The condition of T8.3 is then an envelope of a truth which was left in V as the result of the treatment of procedure statements (explained below). The block number of the smallest embracing block of the procedure statement occurs as "<bnl>" in this condition. This is not the same as the block number at the moment that "<idl><bcsl>" is evaluated, since the procedure call mechanism will have changed the block number.

Evaluation of the right part of T8.3 has the following effect:

a. Evaluation of the first simple name stores the current block number in such a way that it can be reset later (see d).

b. The block number of the block in which the procedure statement occurs is added to V (hence, this becomes for the moment the current block number).

c. "<expl>" is evaluated and a rule which contains this result is added to V.

d. The block number which was preserved in a is reset.

e. Now the value of "<idl><bcsl>" is the value of "result".

Remark: The manipulations with the block number are necessary to avoid clash of names, e.g. in the following case:

```
begin procedure P(f);
      begin integer a; ... f ... end P;
      integer a;
      ...; P(a); ...
end
```

When neither T8.4 nor T8.3 is applicable, and if we assume that "<idl>" is not a function designator (this case is treated below), then by T8.2 the value of "<idl><bcs2><bcl>" (where "<bcsl>" = "<bcs2><bcl>") is the value of "<idl><bcs2>"; i.e., the smallest embracing block is searched. (This is the same technique as was used in the check in phase 2 whether all identifiers are declared.)

Again the three possibilities are considered, viz.

a. A value was assigned dynamically to "<idl><bcs2>".

b. "<idl><bcs2>" was declared in the block with block number "<bcs2>", but no assignment occurred (T8.4).

c. "<idl><bcs2>" is a formal parameter (T8.3).

In case of no success, T8.2 is applied again, etc.

Eventually, this process must come to an end, since "<idl>" is certainly a declared identifier or a formal parameter (this was checked already in phase 2); thus, there will be some block, embracing the initially considered one, in which one of the three above mentioned possibilities holds.

6.16. Array declarations (T9.1 to T9.29)

T9.1 to T9.9 give the syntactical definition of an array declaration.
T9.10 to T9.13 define the value of an array declaration in phase 1.
The meaning of T9.10 is clear. Application of T9.11 has the following effect:

a. The first simple name of its right part is evaluated by application of either T9.13 or T9.12. If T9.13 is applicable, then "<idl>" has been declared already in the same block and "ω" is added to V. Otherwise, a truth is added to V, containing the type of the identifier, an indication that it is an array identifier, and the block number of the block in which it is declared.
b. The three remaining simple names check whether the identifiers in the bound pair list have been declared. This check is performed in phase 1, since these identifiers must have been declared in embracing blocks. First the block number of the smallest embracing block is activated, then "<bplistl> in <decl bplist>" is evaluated, and finally the block number of the block concerned is restored. (By the definition of the program point, the block number of the smallest embracing block is immediately available.)

T9.14 defines the value of an array declaration in phase 2 to be "tr".
T9.15 to T9.29 define the value of an array declaration in phase 4:

a. By T9.19 and T9.20, an integer bound pair list is defined as a bound pair list which contains only integers.
b. By T9.15, if the array declaration contains a bound pair list which is not an integer bound pair list, the expressions in the bound pair

list are evaluated, again after first activating the block number of the smallest embracing block, and later on reactivating the block number of the current block (T9.16 to T9.18, T9.21).

c. The treatment of a non own array declaration is completed by T9.22, T9.24 and T9.25. Eventually, a truth is added to V, containing the identifier concerned, its type, the indication "array", and the evaluated bound pair list.

d. The value of an own array declaration is given by T9.23 to T9.26. Essentially, the same scheme is used as with own simple variables. Only one extra difficulty arises: According to [38], 5.2.5, when a subscripted variable is evaluated, which corresponds to an own array and which has obtained a value in a former activation of the block concerned, it is necessary to check whether the subscripts are within the most recently calculated subscript bounds. This is accomplished by the condition in the second metaexpression of the right part of T9.26: Only if the subscripts are within the most recently calculated subscript bounds (i.e. if the value of "<sub exp list> within bounds of <int bplist1>" (defined in T9.27 to T9.29) is tr) is the value of the subscripted variable "<id1><bcs1>[<sub exp list1>]" equal to the value of the same variable in the previous activation, viz. "<id1><bcs2>[<sub exp list1>]".

e. The meaning of T9.27 to T9.29 is obvious.

6.17. The value of a subscripted variable (T10.1 to T10.9)

T10.1 to T10.4 define the value of a subscript expression list. If a subscripted variable, say"<id1>[<sub exp list1>]",is evaluated, T10.5 results in the evaluation of "<sub exp list1>" and the extension of "<id1>" with the current block number. T10.9 will be applicable to the result, if no assignment has been made to the subscripted variable in the block in which it has been declared, whence its value is undefined. T10.8 gives the replacement of the formal array identifier "<id1><bcs1>" by the actual array identifier "<id2><bcs2>". Note again that"<bcs2>" is the block number of the block in which the procedure statement occurs. T10.7 causes the search in an embracing block.
T10.6 is applicable if none of the aforementioned cases occurs.

6.18. Switch declarations (T11.1 to T11.13)

T11.1 to T11.8 need no further explanation.
T11.9 to T11.13 define the value of a switch declaration in phase 4. We demonstrate the effect of these truths by an example: Evaluation of
"switch S := L, if i > 0 then P else Q, M[3] <p> 4"
in a block with block number "<bcs1><dbcs1>"leads to addition to V of:
switch S <bcs1><dbcs1> co
S <bcs1> [1] eq L co
S <bcs2> [2] eq if i > 0 then P else Q co
S <bcs2> [3] eq M[3]

Remark: "<ui>" in T11.12 and T11.13 stands for "unsigned integer", and is defined in T22.59. The definition of addition is also given later.

6.19. "Label declarations" (T12.1 to T12.4)

T12.1 and T12.2 have the usual meaning.
By means of T12.3 and T12.4, the evaluation of "<label1><pl> 3" results in the addition to V of a truth which contains "<label1>", the current block number and for counter (see section 6.23), and the program point corresponding to the statement which is labelled by "<label1>".

6.20. Procedure declarations (T13.1 to T13.31)

T13.1 to T13.11 define the syntax of a procedure declaration.
T13.12 and T13.13 define the value of a procedure declaration in phase 1. If the procedure identifier has not been declared before in the same block, a truth is added to V containing the identifier, an indication that it is a procedure identifier, possibly of some type, the current block number, and possibly a formal parameter part. The addition of the formal parameter part is used later to check in phase 2 whether the number of actual parameters in a procedure statement is equal to the number of formal parameters in the corresponding declaration.
T13.14 to T13.19 define the value of a procedure declaration in phase 2. We explain only T13.16, the others being similar. Let
"procedure <id1> (<id list1>); <value part1><spec part1><st1><pl> 2"
be the declaration concerned.

a. An extra "begin" is evaluated (to ensure the right scope for the formal parameters).

b. "formal <id list1>, <p1> κ" is evaluated. "<p1> κ" is the extra formal parameter which was already mentioned in 6.4. The effect of evaluating a list of formal parameters is the addition to V of these parameters, supplied with the current block number and the indication "formal" (T13.18, T13.19). This information is used later on in phase 2, in the check whether the procedure body contains only declared identifiers.

c. "begin integer dummy; <stl>; goto <p1> κ end in <decl block>" is evaluated. This means that the prescan mechanism is activated (T5.14) for the procedure body. Note that "<stl>" is embedded in an auxiliary block (by means of the declaration "integer dummy") and that an extra goto statement is inserted, leading to the extra formal parameter "<p1> κ".

d. The first program point of the procedure body is stored by applying T13.20 (cf. T5.12).

e. The "end", corresponding to the "begin" in a, is evaluated.

Remark: From T13.15 and T13.17 it follows that no extra formal parameter is inserted for type procedures.

T13.21 to T13.31 define the value of a procedure declaration in phase 4. By applying T13.21, the procedure declaration is first extended with the first program point of its body. This first program point is available as a result of one of T13.14 to T13.17, and T13.20.
Next the extra formal parameter is added in case of a non type procedure (T13.22, T13.23). In the left parts of T13.22 and T13.23, "<p1>" is the program point corresponding to the procedure declaration, and "<p2>" the first program point of its body. Once the addition of the extra formal parameter is made, "<p1>" is no longer necessary and is therefore omitted. This omission is also done in case of type procedures by T13.24.
T13.25 to T13.28 define some auxiliary metavariables. After application of T13.22 to T13.24 two possibilities arise:

a. The procedure declaration has no value part. Then by T13.29, the relevant information is added to V. Note that the specification part is ignored.

b. The procedure declaration does have a value part. Then by T13.30 and T13.31, the entries in the formal parameter list which occur in the value part are supplied with a special indication, viz. the corresponding specifier. If this process is completed for all value parameters, T13.29 will be applicable.

6.21. Assignment statements (T14.1 to T14.44)

T14.1 to T14.16 give the syntax of assignment statements and of declared assignment statements.

T14.17 defines the value of an assignment statement in phase 2.

T14.18 links the assignment statement with its successor.

T14.19 to T14.44 define the value of an assignment statement in phase 4. The ultimate result of the application of these truths to an assignment statement is the addition to V of: the variable concerned, followed by the block number of the block in which it has been declared, followed by "is", followed by the expression on the right hand side of the assignment statement (cf. T14.41 to T14.44).

Complications in the detailed definition of the evaluation of an assignment statement are caused by:

a. Multiple assignment statements. The requirement that the expression on the right hand side is evaluated only once does not allow the first solution which comes to mind, i.e., the rewriting of the multiple assignment statement as a sequence of "simple" assignment statements.
b. The desire to supply the variables of the left part list with the block number of the block in which they are declared (and not of the block in which the assignment statement occurs).
c. Clash of names, especially in the case of assignment to a formal parameter which has a subscripted variable as its corresponding actual parameter.
d. Assignment to the procedure identifier in the declaration of a type procedure.

e. The requirement that subscripted variables in a left part list have subscripts within the corresponding array bounds.

The first two problems are solved essentially by means of the introduction of the auxiliary metavariables "<ext left part>" and "<ext left part list>", and the usual search in embracing blocks. Here a scheme is used which first establishes the identity of the variables in the left part list, and then evaluates the expression on the right hand side, after which the rewriting of the assignment statement as a sequence of "simple" assignment statements becomes possible. Then T14.41 to T14.44 become applicable.

Clash of names is treated by T14.32 and T14.33. The structure of T14.32 is similar to that of T8.3.

Assignment to a procedure identifier is defined in T14.34. It will be explained later when we treat type procedures.

The check whether the subscripts of a subscripted variable are within the corresponding subscript bounds is performed by evaluating the first simple name of the right part of T14.40. The value of this simple name was defined in T9.27 to T9.29. If it has the value "<u>tr</u>", it will be added to V, again without any influence on the evaluation of the remainder of the program. However, if its value is not "<u>tr</u>", then "ω" will be added to V with the usual result (6.7).

6.22. <u>Goto statements (T15.1 to T15.19)</u>

T15.1 defines the value of a goto statement in phase 2 and T15.2 to T15.19 define its value in phase 4.

The requirement that a goto statement, leading to an undefined switch designator, be equivalent to the dummy statement has complicated the definition of the goto statement, among other things because it is necessary to keep available the program point corresponding to the goto statement concerned, and the block number of the block in which this statement occurs.

By T15.2, the current block number is added to the goto statement.

By T15.3, parentheses around designational expressions are deleted.

T15.4 to T15.6 treat conditional designational expressions. If the boolean expression of the if clause is not one of the symbols "true" or "false", then this boolean expression is evaluated by T15.4, after which T15.5 or T15.6 may apply (cf. also 4.2.3.1).

After application of T15.3 to T15.6, the designational expression is either a label or a switch designator.

T15.7 to T15.12 treat the first case.

By T15.7, the current for counter is found and added to "goto <label1> <p1><bn1>". The definition of the for counter, as given in the section on for statements, is essentially similar to that of the block number: Possible for counters are: "φχ", "φχφχφφχ", "φχφφχφχφφφχ", etc. Again, the "χ"'s count the depth of nested for statements, the "φ"'s between a certain "χ" and the immediately preceding "χ" count the number of parallel for statements on the depth of this "χ". The for counter is used to avoid jumps into a for statement from outside (see below).

By T15.8, the current block number is added to "goto <label1><fgs1> <p1><bn1>". Note that, although at the moment of application of T15.2, "<bn1>" was the current block number, this is now no longer necessarily the case, since the block number may have changed as the result of the treatment of switches (see explanation of T15.17 below).

T15.10 defines the usual transition to a search in the embracing block, and T15.9 applies if the outermost block is reached.

T15.11 treats the case of a formal label: this label is replaced by the corresponding actual designational expression. First, however, the block number in which the procedure statement containing the formal label occurs, is activated, in order to avoid clash of names. It is not necessary to reactivate the current block number, since this will be activated eventually by T15.12.

If "<label1>" occurs in the block with block number "<bcs1><dbcs1>", then T15.12 will be applicable to "goto <label1><bcs1><fgs1><fgs><p><bn>".

The following remarks may explain T15.12:

a. As a result of application of T12.3 and T12.4, a truth will have been left in V which is enveloped by the condition of T15.12.

b. "<bcsl><dbcsl>" is the block number of the block in which "<labell>" was "declared". It is activated by the evaluation of the first simple name of the right part of T15.12.

c. "τ <pl>" is the program point, corresponding to the statement which is labelled by "<labell>". Evaluation of this program point in the right part of T15.12 leads to the continuation of the evaluation of the program by evaluating this statement.

d. "<fgsl>" is the for counter, current at the moment of "declaration" of "<labell>". It is activated by the evaluation of the second simple name of the right part of T15.12. If, at the moment that "goto <labell> <bcsl><fgsl><fgs><p><bn>" is evaluated, the for counter does not have the form "<fgsl><fgs>", then T15.12 will not be applicable: From the definition of the for counter it follows that jumps into a for statement from outside are prevented (in the sense that then only T0.1 will be applicable).

e. The program point corresponding to the goto statement concerned and the block number of the block in which this statement occurs (the last two metavariables in the left part of T15.12) have no function in T15.12.

T15.13 to T15.19 define the value of a goto statement in the case that the designational expression is a switch designator.
By T15.13, the switch identifier is extended with the current block number and the subscript of the switch designator is evaluated.
T15.14 to T15.16 have the usual function.
If T15.17 is applicable then first the block number of the block in which the switch concerned is declared is added to V. Again, this is done to avoid clash of names ([38], 5.3.5). The second simple name of the right part of T15.17 is evaluated by application of T15.18 or T15.19.
T15.19 will be applicable if the value of the subscript in the switch designator is equal to the ordinal number of one of the items in the corresponding switch list. Then, as a result of the treatment of switch declarations, a truth will have been added to V which is enveloped by the condition of T15.19, and the evaluation of the original goto statement will be replaced by the evaluation of the goto statement leading to

the corresponding designational expression in the switch list.
If, on the other hand, T15.19 is not applicable, then by T15.18 the evaluation of the goto statement concerned will simply be replaced by the evaluation of "τ <pl> α", i.e., of its successor. Note that first the block number of the block in which this goto statement occurs is reactivated; this block number was added to the goto statement by T15.2. Thus, a goto statement, leading to an undefined switch designator, is equivalent to the dummy statement (apart from side effects in the evaluation of the subscript; cf. also 6.1).

6.23. For statements (T16.1 to T16.32)

T16.1 to T16.7 define the syntax of a for statement.
T16.8 and T16.9 give the definition of the for counter. As will be seen from T16.13, an auxiliary terminal symbol "forbegin" is evaluated in phase 3 of the evaluation of a for statement. The value of this symbol is defined in T16.10, T16.11 and T16.12. These truths are analogous to T6.1, T6.3 and T6.4, respectively. Together with the truths defining the value of "forend" (given later), they perform the updating of the for counter.
The prescan rules for the for statement are given in the rest of section 16. The main reason for their complex structure is the fact that it is not correct to rewrite a for statement, containing a for list with more than one element, as a sequence of for statements, each containing just one element of this for list (thus, the proposed semantics of the for statement in [24] contains an error). This was pointed out to us by B.J. Mailloux and is demonstrated by the following example:
"for i := 1, 2 do begin own integer j; if i = 1 then j := 0; j := j + 1 end"
is not equivalent with:
"for i := 1 do begin own integer j; if i = 1 then j := 0; j := j + 1 end;
for i := 2 do begin own integer j; if i = 1 then j := 0; j := j + 1 end".

The essential feature in the prescan rules for the for statement is the introduction of a "dynamic label", called "special label <pl>". Here we mean by "dynamic" that this special label is associated successively with different labels in the program (also especially introduced for this purpose). It is then possible, after completion of an evaluation of the statement after the for clause, to resume the evaluation of the for statement with the next assignment to the controlled variable, by means of a jump backwards to this dynamic label.

A precise description now follows:

First we consider T16.13. Its right part consists of two simple names. The structure of its first simple name is similar to that used in the prescan rules of sections 4 and 5 of the metaprogram. Apparently, its only use is the evaluation of "forbegin" in phase 3. In the second simple name we observe:

a. The introduction of the extra labels "<pl> μ 1" and "<pl> μ 2".
   The label "<pl> μ 1" labels the statement "<st1>" that occurs after the for clause. In the remainder of this section, we shall call "<st1>" the "controlled statement".
   The label "<pl> μ 2" labels the construction "forend (<int var1>)". This construction will be used later at the end of the evaluation of the for statement. Note that "forend (<int var1>)" has syntactically the form of a procedure statement, since "forend" is an auxiliary identifier (T23.198).
b. The introduction of the extra goto statement "goto special label <pl>". Again, this is a syntactically correct goto statement, since "special label <pl>" is an auxiliary label by T23.205.
c. From a and b it follows that the sequence of symbols after "<pl> μ 1" is indeed a blockend. This fact is used later, in the left parts of T16.14 to T16.20.

After application of T16.13, one of T16.14 to T16.19 will be applicable to the second simple name of the right part of T16.13. T16.14 to T16.16 treat the case in which the for list contains more than one element, and T16.17 to T16.19 the other case. Suppose T16.14 is applicable. The first simple name of its right part has the usual structure. We see that, in phase 4, a correspondence is set up between the special label and the auxiliary label "<pl> μ 3".

From the second simple name it follows that:

a. After execution of "<int var1> := <aexp1>", a jump is performed to the controlled statement ("<p2> μ 1" labels this controlled statement as a result of T16.13; note that "<p2>" is fixed for the whole for statement).

b. After execution of the controlled statement, "goto special label <p1>" will be executed. As a result of the association of the special label and "<p1> μ 3", this jump will cause the next assignment to the controlled variable to be executed.

In T16.15 the same principle is used. The jump to the controlled statement is executed only if the boolean expression after "while" has the value "true"; otherwise, the next element of the for list is considered.
T16.16 defines the step-until element. The second simple name of its right part is similar to [38], 4.6.4.2. E.g., "<p1> μ 7" corresponds to the label "Element exhausted", and "<p1> μ 6" to the label "L1". For the definition of the integer procedure "sign", see T2.1.
T16.17 to T16.19 treat for list elements, in the case that these elements are the last ones of the for list.
In T16.17, the special label is now associated with "<p2> μ 2"; by T16.13, this label labels the construction which ends the for statement. Hence, "goto special label <p2>" will here cause the evaluation of "forend (<int var1>)".
T16.18 and T16.19 are similar to T16.16 and T16.17, but now "<p2> μ 2" corresponds to the label "Element exhausted".
T16.20 gives the prescan rule for the auxiliary statement "goto special label <p1>". From its structure, it follows that only phase 4, in which T16.21 will be applicable, is of importance. We see that the jump to the special label is replaced by a jump to the auxiliary label most recently associated with it: as a result of one of T16.14 to T16.19, a truth will have been left in V which is enveloped by the condition of T16.21.
T16.22 is the prescan rule for the end of the for statement. The requirement that the value of the controlled variable be undefined upon exit from the for statement makes the remaining truths of this section necessary. First, by evaluating the first simple name of the right part of T16.23,

the for counter is updated. Next, the value of the controlled variable is set to "ω". The usual technique for the search in embracing blocks and the treatment of formal parameters is applied (cf. e.g. T16.27 with T14.32). Ultimately, either T16.28 or T16.32 will apply, resulting in the addition to V of a truth which defines the value of the controlled variable to be "ω".

6.24. Procedure statements and function designators (T17.1 to T17.104)

T17.1 to T17.61 define:
"<proc id>" and "<decl proc id>",
"<int proc id>" and "<decl int proc id>",
"<boolean proc id>" and "<decl boolean proc id>",
"<proc st>" and "<decl proc st>",
"<int funct des>" and "<decl int funct des>",
"<boolean funct des>" and "<decl boolean funct des>",
"<act par>" and "<decl act par>", and
"<act par list>" and "<decl act par list>".

The mechanism explained in 6.5 is used extensively. In the cases of "<decl proc st>", "<decl int funct des>", and "<decl boolean funct des>", it is checked whether the number of actual parameters is equal to the number of formal parameters in the corresponding declaration (T17.57 to T17.59 and T17.60, T17.61).

T17.62 gives the value of a procedure statement in phase 2. The remaining truths of this section treat procedure statements and function designators in phase 4.

After application of T17.63, T17.64,(T17.66), T17.68 and T17.70, which have the usual meaning, to a procedure statement (supposing that the procedure concerned is not a function designator), either T17.72 or T17.73 will prove to be applicable.

A similar scheme is used for function designators in T17.65, T17.67, T17.69 and T17.71, after which T17.74 will be applicable. Note, however, the differences between the two cases: A procedure statement is always accompanied by its corresponding program point (e.g. "<pl>" in T17.63), which clearly does not exist for function designators. Also, an empty actual parameter part does not occur here with function designators, since this case will be taken care of by T8.1 to T8.3.

T17.72 treats procedure statements without parameters. Evaluation of its right part has the following effect:

a. The block number of the block in which the corresponding declaration occurs is activated by evaluating "enter procedure <idl><bcsl>" (cf. T17.77 and section 6.3).
b. The extra formal parameter "<p3> κ" is associated (see below) with the extra actual parameter "<p2> κ". Cf. also T5.7.
c. The first program point of the procedure is evaluated.

One might expect the evaluation of "exit procedure" as the fourth simple name, corresponding to the "enter procedure" of a. However, this is not necessary, since the correct block number is activated after the completion of the evaluation of the procedure statement as a result of the evaluation of the inserted auxiliary goto statement (this fact was ignored in the second example of section 6.3 on block numbers).

A procedure statement with parameters is evaluated by means of T17.73. Again, the procedure entrance is performed, and the formal parameters (which include the extra formal label) are associated with the actual parameters, after which the first program point of the procedure body is evaluated.

T17.74 treats function designators. The following simple names are evaluated:

a. "enter procedure <idl><bcsl>". The procedure entrance is performed.
b. An extra "begin", in view of:
c. "<typel><idl> π". This is a type declaration; hence, T7.11 will be applicable (cf. also T23.199). It is introduced to make assignment to the procedure identifier possible (T14.34). The extra symbol "π" is necessary in recursive situations. Without this indication, an occurrence of the procedure identifier other than as a left part, would not cause recursive activation of the procedure, but would simply deliver the value that was last assigned to it.

d. "<ext formal par part1> substitute <act par part1>". The formal parameters are associated with the actual parameters. Cf. also T17.104.
e. "<pl>". The first program point of the procedure body is evaluated.
f. "function value : va {<idl> π}". The value assigned to the procedure identifier is stored. Cf. T17.79.
g. "exit procedure". The block number of the block in which the function designator occurs is restored (T17.78). By the definition of "exit procedure", it is not necessary to include the evaluation of an "end", corresponding to the "begin" of b.
h. "function value". Thus, finally, the value of the function designator is the value of "function value", as stored by T17.79.

T17.75 and T17.76 treat function designators, occurring as statements. These cannot be treated as "normal" procedures, since no extra goto statement was included at the moment of their declaration. The solution to this difficulty is provided by including such a function designator in an auxiliary assignment statement. Note that the left parts of these assignment statements have been declared in T2.1. The correct sequencing is ensured here by evaluating "τ <pl> α", which corresponds to the successor of the procedure statement.
T17.80 to T17.104 define the formal-actual substitution.
T17.81 defines the call by value of a formal parameter, specified "integer" or "boolean":
a. An extra "begin" is evaluated.
b. The formal parameter is declared to be of the specified type.
c. The assignment to the formal parameter is performed (T17.82), with some precautions because of the possibility of clash of names: Before the evaluation of the assignment statement, the block number of the block in which the procedure statement or function designator occurs is activated.
d. The formal-actual substitution of the remaining parameters is performed, if necessary. Cf. also T17.103.

Again, no "end" corresponding to the "begin" of a is evaluated. The correct block number will be activated upon exit from the procedure either by the extra goto statement in case of procedure statements, or by the evaluation of "exit procedure" in case of function designators.

By T17.83, a formal parameter which was called by value and specified "integer procedure" or "boolean procedure", is treated as a formal parameter, called by value and specified "integer" or "boolean".

T17.84 to T17.97 treat value arrays.
By T17.84, first an extra "begin" is evaluated, then follows the evaluation of "<type1> array <id1> actual <id2>" (see below), after which the remaining substitutions are performed, if necessary.
By T17.85 to T17.89, the declaration of the actual array identifier is looked up, after which the formal identifier is declared to be an array with the same bounds as the actual (first simple name of the right part of T17.89). The evaluation of the second simple name of the right part of T17.89 will result in the assignment of the value of the actual array (i.e. of the ordered set of values of the corresponding array of subscripted variables, [38], 2.8) to the newly declared array. This assignment is performed by application of T17.91 to T17.95. Auxiliary truths for this purpose are T17.90, T17.96 and T17.97.

Finally, we explain the treatment of formal parameters called by name. If there are formal parameters left in the extended formal list which are called by value, they are treated first (T17.98); otherwise, T17.100 is applicable. First an extra "begin" is evaluated to ensure the correct scope of the formal parameter. The second simple name of the right part of T17.100 is evaluated by application of T17.101, resulting in the addition to V of a truth containing the formal parameter, the block number of the block which was entered in T17.100, the corresponding actual parameters, and the block number of the block in which the procedure statement or function designator occurs. The use of such a truth was already demonstrated in T8.3, T10.8, T14.31, T14.32, etc. The section ends with the auxiliary truths T17.102 to T17.104.

6.25. Variables (T18.1 to T18.44)

In this section, the definitions of variables and of declared variables are given. The technique described in 6.5 is used.

6.26. Syntax of arithmetic expressions (T19.1 to T19.30)

This section is simply a transcription of [38], 3.3.1, together with the definition of the "declared" counterparts of the metavariables concerned.

6.27. Syntax of boolean expressions (T20.1 to T20.40)

See section 6.26.

6.28. Syntax of designational expressions (T21.1 to T21.25)

See section 6.26.

6.29. The value of boolean expressions and of arithmetic expressions (T22.1 to T22.92)

By T22.1 and T22.2, the value of an expression between parentheses is equal to the value of the same expression with the parentheses deleted. T22.3, T22.5 and T22.7 define the value of a conditional arithmetic expression. If the boolean expression of the if clause is not one of the symbols "true" or "false", it will be evaluated by application of T22.3, after which T22.5 or T22.7 may be applicable (cf. also 4.2.3.1). Similarly, the value of a conditional boolean expression is defined in T22.4, T22.6 and T22.8.
T22.9 to T22.14 give the value of a simple boolean expression, which is neither a boolean primary different from a relation, nor is enveloped by one of the left parts of T22.15 to T22.41. Cf. also 4.2.3.2.
Note that the observance of the precedence rules for the operators is achieved by the definition of T22.9 to T22.14. For the sake of completeness, we mention the relevant truths for evaluation of a boolean primary:
a. The value of a logical value is itself (T23.22).

b. A boolean variable is evaluated by means of T8.1 or T10.5.

c. The value of a relation is given by T22.9.

d. The value of "(<bexp>)" is given by T22.2.

e. The value of a boolean function designator is given by T8.1 or T17.65.

T22.15 to T22.32 define the usual truth tables for the operators "¬", "∧", "∨", "⊐", and "≡".

T22.33 to T22.41 define relations involving integers; every relation is first reduced to the relation "<int> ≤ <int>", which is in turn reduced to the evaluation of "<int> ≤ 0", as defined in T22.39 to T22.41.

T22.42 and T22.43 are used in the evaluation of expressions as "+(-3)", and T22.44 to T22.46 in the evaluation of e.g. "3+(-5)" and "+3 ≤ +5" (by T22.38, this leads to the evaluation of "+3 - +5").

T22.47 to T22.50 define the value of a simple arithmetic expression involving integer variables or function designators or containing more than one operator. Again, the precedence of the operators is observed in these truths. Note the deviant form of the right part of T22.47 (the value of "(-2) ↑ 2" is not equal to the value of "-2 ↑ 2").

T22.51 to T22.53 define exponentiation. Since exponentiation is not defined for non-positive exponents (this would lead to "real" numbers), the value of the expression after "else" in the right part of T22.52 is "ω"; i.e., the value of "0 ↑ 0" is "ω".

T22.54 and T22.55 define integer division. The left part of T22.54 might, for example, be an envelope of the result of application of T22.48 and T22.1 to "3 ÷ (-5)".

T22.56 to T22.58 define multiplication.

T22.59 to T22.61 define the syntax of an unsigned integer, an integer and a sequence of zeroes.

T22.62 to T22.92 define addition and subtraction of integers (cf. [49], p. 17, 18 and 4.2.3.3).

6.30. Basic symbols and auxiliary symbols. Comment conventions (T23.1 to T23.217)

T23.1 to T23.15 define the syntax of identifiers, constants (cf. T8.6, T14.26, etc.) and digits.

T23.16 to T23.19 define the value of a number.

T23.20 to T23.22 define the syntax and the value of a logical value.

T22.23 to T23.74 list the letters.

By T23.75 to T23.119, an "end comment symbol" is any ALGOL 60 basic symbol except the symbols ";", "end" and "else". By T23.122 to T23.124, a "comment symbol" is any ALGOL 60 basic symbol other than a semicolon. These definitions are used to define the comment conventions of [38], 2.3 in T23.135 to T23.137.

By T23.134 a parameter delimiter which is not a comma is replaced by a comma.

T23.138 to T23.141 are introduced because of [38], 3.5.5.

T23.142 to T23.193 list the auxiliary symbols which were introduced in the preceding sections.

T23.194 to T23.200 list the auxiliary identifiers; the auxiliary labels are given by T23.202 to T23.205.

T23.207 to T23.209 define a sequence of basic and auxiliary terminal symbols, used in T0.1, T6.5 and T23.210.

The remaining truths were explained in 6.7.

BIBLIOGRAPHY

1. American Standards Association Subcommittee X3.4.2 — Suggestions on ALGOL 60 (Rome) Issues. Comm. ACM, 1963, vol. 6, pp. 20-23.

2. J.W. Backus — The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference. ICIP Proceedings, Paris, 1959. London, Butterworth's, 1960, pp. 125-132.

3. J.W. de Bakker — Formal definition of algorithmic languages, with an application to the definition of ALGOL 60. Report MR 74, Mathematisch Centrum, Amsterdam, 1965.

4. C. Böhm — The CUCH as a formal and description language. [41], pp. 179-197.

5. C. Böhm — Introduction to CUCH. Automata Theory (Ed. E.R. Caianiello). New York, Academic Press, 1966.

6. A. Caracciolo di Forino — On the concept of formal linguistic systems. [41], pp. 37-51.

7. A. Caracciolo di Forino — Generalized Markov Algorithms and Automata. Automata Theory (Ed. E.R. Caianiello). New York, Academic Press, 1966.

8. A. Caracciolo di Forino — String processing languages and generalized Markov algorithms. Proc. IFIP Working Conference on Symbol Manipulation Languages, Ed. D. Bobrow. (to appear).

9. A. Caracciolo di Forino, L. Spanedda, N. Wolkenstein — PANON IB, a programming language for symbol manipulation. Presented at the SICSAM Symposium, Washington, March 29-31, 1966.

10. N. Chomsky — Three models for the description of language. IRE Transactions on Information Theory, 1956, vol. IT-3, pp. 113-124.

11. N. Chomsky — On certain formal properties of grammars. Information and Control, 1959, vol. 2, pp. 137-167.

12. N. Chomsky — Context free grammars and pushdown storage. Quarterly Progress Report no. 65. Research Laboratory of Electronics, M.1.T, 1962.

13. N. Chomsky and G.A. Miller — Finite state languages. Information and Control, 1958, vol. 1, pp. 91-112.

14. C. Christensen — Examples of symbol manipulation in the AMBIT programming language. ACM National Conference Proc., August 1965, pp. 247-261.

15. K. Cohen and J.H. Wegstein — AXLE, an axiomatic language for string transformation. Comm. ACM, 1965, vol. 8, pp. 657-661.

16. M. Davis — Computability and Unsolvability. New York, McGraw-Hill,, 1958.

17. C.C. Elgot — Machine species and their computation languages. [41], pp. 160-179.

18. C.C. Elgot and A. Robinson — Random-access, stored program machines, an approach to programming languages. J. ACM, 1964, vol.11, pp. 365-399.

19. J. Feldman — A formal semantics for computer languages and its application in a compiler-compiler. Comm. ACM, 1966, vol. 9, pp. 3-9.

20. R.W. Floyd — On the non-existence of a phrase structure grammar for ALGOL 60. Comm. ACM, 1962, vol. 5, pp. 483-484.

21. J.V. Garwick — The definition of programming languages by their compilers. [41], pp. 139-147.

22. S. Ginsburg — The mathematical theory of context free languages. New York, McGraw-Hill, 1966.

23. S. Ginsburg and H.G. Rice — Two families of languages related to ALGOL. J. ACM, 1962, vol. 9, pp. 350-371.

24. C.A.R. Hoare — Cleaning up the for statement. ALGOL Bulletin, no. 21, pp. 32-35.

25. S. Igarashi — A formalization of the description of languages and the related problems in a Gentzen type formal system. RAAG Research Notes, Third Series, no. 80, 1964.

26. S. Igarashi — An axiomatic approach to the equivalence problems of algorithms with applications. Ph.D. thesis, University of Tokyo, 1964.

27. D.E. Knuth — A list of the remaining trouble spots in ALGOL 60. ALGOL Bulletin, no. 19, pp. 29-38.

28. F.E.J. Kruseman Aretz — ALGOL 60 Translation for Everybody. Elektr. Datenverarbeitung, Heft 6, 1964, pp. 233-244.

29. S.Y. Kuroda — Classes of languages and linear bounded automata.
Information and Control, 1964, vol. 7, pp. 207-223.

30. P.J. Landin — The mechanical evaluation of expressions.
Comp. J., 1964, vol. 6, pp. 308-320.

31. P.J. Landin — A formal description of ALGOL 60.
[41], pp. 266-294.

32. P.J. Landin — A correspondence between ALGOL 60 and Church's lambda notation.
Comm. ACM, 1965, vol. 8, pp. 89-101, pp. 158-165.

33. A.A. Markov — Theory of Algorithms.
Moscow, USSR Academy of Sciences, 1954. (Translated into English by the Israeli Program for Scientific Translations, Jerusalem, 1961).

34. J. McCarthy — A formal description of a subset of ALGOL.
[41], pp. 1-12.

35. J. McCarthy — Problems in the theory of computation.
Proc. IFIP Congres 1965, vol. 1 (ed. A. Kalenich), Washington, Spartan Books, 1965, pp. 219-222.

36. J. McCarthy et.al. — The LISP 1.5 programmer's manual.
Cambridge, Mass. MIT Computation Center, 1962.

37. E. Mendelson — Introduction to Mathematical Logic.
Princeton, van Nostrand, 1964.

38. P. Naur (Ed.) — Revised Report on the algorithmic language ALGOL 60.
Copenhagen, Regnecentralen, 1962.

39. M. Nivat and N. Nolin — Contribution to the definition of ALGOL semantics.
[41], pp. 148-159.

40. T.B. Steel, jr. — Beginnings of a theory of information handling.
Comm. ACM, 1964, vol. 7, pp. 97-103.

41. T.B. Steel, jr. (Ed.) — Formal Language Description Languages for Computer Programming.
Proceedings IFIP Working Conference, Vienna, 1964.
Amsterdam, North-Holland, 1966.

42. T.B. Steel, jr. — A formalization of semantics for programming language description.
[41], pp. 25-36.

43. C. Strachey — Towards a formal semantics.
[41], pp. 198-220.

44. H. Thiele — Wissenschaftstheoretische Untersuchungen in Algorithmische Sprachen.
Berlin, VEB, 1966.

45. H. Wang — Towards Mechanical Mathematics.
IBM J. of Research and Development, 1960, vol. 4, pp. 2-22.

46. N. Wirth — A generalization of ALGOL.
Comm. ACM, 1963, vol. 6, pp. 547-554.

47. N. Wirth and H. Weber — EULER, a Generalization of ALGOL, and its Formal Definition.
Comm. ACM, 1966, vol. 9, pp. 13-23, pp. 89-99.

48. A. van Wijngaarden — Generalized ALGOL.
Proc. ICC Symposium on Symbolic Languages in Data Processing.
New York, Gordon and Breach, 1962, pp. 409-419.
Also in
Annual Review in Automatic Programming, R. Goodman (Ed.), vol. 3, pp. 17-26.
New York, Pergamon Press, 1963.

49. A. van Wijngaarden — Recursive definition of syntax and semantics.
[41], pp. 13-24.

50. A. van Wijngaarden — Orthogonal design and description of a formal language.
Report MR 76, Mathematisch Centrum, Amsterdam, 1965.

51. Y.I. Yanov — The logical schemes of algorithms.
Problems of Cybernetics, vol. 1, pp. 82-140.
New York, Pergamon Press, 1960.

52. V. Yngve — An introduction to Comit programming.
M.I.T., 1961.

53. PL/I-Definition Group of the Vienna Laboratory — Formal Definition of PL/I.
IBM Technical Report TR25.071, December 1966.