STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

REKENAFDELING

MR 74

FORMAL DEFINITION OF ALGORITHMIC LANGUAGES

with an application to the definition of ALGOL 60

J.W. de Bakker

mei 1965

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

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

TABLE OF CONTENTS

Chapter 1

1. Introduction

In this preliminary report, we demonstrate the metalanguage, developed by A. van Wijngaarden in [5,6], by means of a formal description of the syntax and semantics of ALGOL 60.
In Chapter 1, section 2, we describe informally the structure of the metalanguage; we intend to complete this definition later on in terms of an ALGOL program. In section 3 we list some simple examples.
In Chapter 3 we give the formal definition of ALGOL 60, preceded in Chapter 2 by an explanation of the techniques used.
In Chapter 4 we define a small selection of the proposals for ALGOL X.

The author is indebted to B.J. Mailloux for many stimulating discussions and for the correction of the English text of this report.

2. The metalanguage

The following, admittedly not unambiguous, description is based on the papers [5,6].

We consider an abstract machine, called the processor P, which reads a text, i.e. any finite sequence of symbols, and produces during the reading another text, called the value of the text so far read. This value is - with a metacomma, denoted by <u>co</u>, as separator - added to a list V of so called truths; this list is constantly at the disposal of the processor: during the evaluation of a sequence of symbols, P scans the list V, examining the truths one by one in order, beginning with the last truth in the list (i.e. the one most recently added to V as the result of the previous evaluation), and sees whether it can apply any one of these truths.

There are several ways in which a truth can be applicable to the evaluation of a sequence of symbols.
In order to explain this, we need the notion of an "envelope". A sequence of symbols τ is an envelope of a sequence of symbols σ in

each of the following three cases:

1. σ and τ are identical.

2. τ is a metalinguistic variable which has the sequence σ as one of its possible values, e.g.

σ : a and τ : <letter> ,

σ : 3+4 and τ : <arithmetic expression>,

σ : <identifier><letter> and τ : <identifier> .

Here we assume that the processor is able to establish the truth of the relations a in <letter> , (this is the equivalence of the definition in BNF: <letter> :: = a), 3+4 in <arithmetic expression> , and <identifier> <letter> in <identifier> , by consulting the list V (see page 8).

3. Suppose τ is a sequence of $n > 1$ symbols and/or metalinguistic variables and suppose σ is also such a sequence of $m \geq n$ elements. Then we try successively all possible partitions of the elements of σ in n subsequences (in a suitably chosen, once and for all fixed, order, see below) and for each of these subsequences see whether 1 or 2 holds. Whenever for one of these subsequences we fail, we try the next partition. If there is no longer a next partition τ is not an envelope of σ.

Example:

σ : ab + ba.

τ : <identifier> + <identifier> .

3.1. We evaluate a in <identifier> . Suppose this has the value true. Then

3.2. We verify whether b and + are identical. They are not, so

3.3. We evaluate ab in <identifier> .

Suppose this has the value true.

3.4. We establish the identity of + and + .

3.5. We evaluate ba in <identifier> (because <identifier> is the only metavariable left we have to take together all the remaining symbols).

The order of partitioning we have chosen is the following:

We partition the m elements of σ into n subsequences of lengths

$m_1, m_2, \ldots m_n$ ($m_i > 0$, $i=1,2,\ldots n$), such that $m_1+m_2+\ldots+m_n = m$. The sequence $m_1, m_2, \ldots m_n$ can be considered as a number N in the base m. Then we perform the partitions defined by increasing N.

We now refine this definition of envelope by extending the notion of metalinguistic variable. A metavariable is denoted by concatenation of the symbol < , a sequence of letters, possibly a sequence of digits, and the symbol > , e.g. <letter1>, <arithmetic expression11>, <identifier> . Here, the function of the digits is that whenever we substitute for a metavariable one of its possible values, we have to perform the same substitutions in all other occurrences of the same metavariable (here we mean by the same: denoted by the same sequence of letters and digits) in the truth concerned, e.g. if we want to know whether <letter1> <letter1> is an envelope of aa, we do the following:

1. We evaluate a <u>in</u> <letter> .
2. We substitute a for the second occurrence of <letter1>.
3. We verify whethera equals a.

Hence, e.g. <digit1> + 0 = <digit1> is not an envelope of 3+0 = 4.

Next we list the four possible cases in which a truth is applicable to the evaluation of a sequence of symbols σ.

1. The truth is an envelope of σ. In this case, the value of σ is defined to be <u>true</u>. E.g. the truth has the form <unsigned integer> ≥0, σ has the form 3 ≥ 0.
2. The truth consists of an envelope of σ, followed by the metasymbol <u>is</u>, followed by some other symbols and/or metavariables. Then the effect of applying this truth to σ is that, instead of evaluating σ , we evaluate the right hand side (i.e. the elements after the <u>is</u> sign), after having made the same substitutions in the right hand side which we made in the left hand side when we established that this left hand side was an envelope of σ. E.g. suppose that σ has the form -3+5, and that the relevant truth has the form

- <unsigned integer1> + <unsigned integer2> <u>is</u> <unsigned integer2>
- <unsigned integer1> ,

then the effect of applying this truth is that we evaluate 5-3 instead (and start all over again; i.e., we try to apply the last truth in V, if this is not applicable the last but one, etc.).

3. The truth consists of an envelope, which we denote for the moment by ε , of σ , preceded by the metasymbol $\rightarrow$, preceded by some other sequence of metavariables and/or symbols, which sequence we denote by λ. We first of all make the same substitutions which we made in ε (to establish that it was an envelope of σ) into the left hand side λ. Then we distinguish the following two subcases:

3.1. There remain no metalinguistic variables in λ. In this case we evaluate λ by exactly the same process as we use for any other sequence of symbols; i.e. we again try to apply the last truth, the one but last, etc. Again there are two possibilities:

3.1.1. We find that λ has the value true. Then we define the value of σ to be true (i.e. now case 3 is reduced to case 1, e.g. we want to know the value of $5 \geq 3$, and we find a truth of the form
<unsigned integer1> $\leq$ <unsigned integer2> $\rightarrow$ <unsigned integer2> $\geq$ <unsigned integer1> .
Suppose we find that true is the value of $3 \leq 5$, then we know that the value of $5 \geq 3$ is also true).

3.1.2. We find that λ has some other value, i.e. any symbol or list of symbols, excluding the symbol true. Then we see whether perhaps there is another partitioning of σ which makes ε into an envelope of σ . If this is the case, we again evaluate λ after the required substitutions (which will in general be different from the substitutions we had to perform the previous time, so it makes sense to evaluate λ anew), and see whether λ now perhaps has the value true. If not, we continue, if possible, with the next partitioning etc. If there is no longer a possible partitioning of σ which makes ε into an envelope of σ, then we have found that we cannot apply the truth concerned to the evaluation of σ .

3.2. λ contains one or more metavariables. Then again we evaluate λ but now we use another method: we see whether λ itself is an envelope of any truth in V (here again we first try the last truth, then the one but last etc.). If this is the case, we define the value of λ and hence also of σ, as true. If not, we try another partitioning of σ

which makes ε an envelope, perform the new substitutions in λ , evaluate λ again by the given special method etc.

4. The truth consists of an envelope, say ε , of σ, followed by <u>is</u>, followed by some other metavariables and/or symbols, which we denote by ρ, while ε is preceded by →, which is in turn preceded by some symbols, say λ, i.e. the truth has the form: λ → ε <u>is</u> ρ .
We proceed analogously to case 3: First we perform the required substitutions in λ.
4.1. If λ contains no metavariables we evaluate it in the normal way. If it has the value <u>true</u> then we also perform the substitutions - which made ε into an envelope of σ - in ρ and continue by evaluating, instead of σ , the resulting right hand side.
If λ has any other value, we try the next partitioning of σ which makes ε into an envelope of σ etc.
4.2. If λ does contain metalinguistic variables, we evaluate it again by scanning V for a truth which is enveloped by λ . If we find such a truth, say τ, we perform both the substitutions which we had to make in order to establish that ε was an envelope of σ and the substitutions which we had to make in order to establish that λ was an envelope of τ, in the right hand side ρ and continue by evaluating the thus modified right hand side.
Example:
The truth concerned is something like:
{<digit2> +1 <u>is</u> <digit1>} → {<digit1> -1 <u>is</u> <digit 2>} , and σ is 6-1. Then we do the following:
a. We establish that <digit1> -1 is an envelope of 6-1.
b. We substitute 6 for <digit1> in λ.
Thus, we now have to evaluate <digit2> +1 <u>is</u> 6.
We find that this still contains a metalinguistic variable, so we see whether it is an envelope of any truth in V.
Suppose we find in V: 5+1 <u>is</u> 6.
c. Then the value of λ is <u>true</u>, and, moreover, we know that we have to substitute 5 for the occurrence of <digit2> in ρ.
d. We continue by evaluating 5.

The metasymbol va may occur in a truth after the metasymbol is or in the sequence of symbols preceding the →.

Example:

The truth has the following form:

<digit1> <plus or minus 1><digit2> is

va {<digit1><plus or minus 1> 1} <plus or minus 1> va {<digit2> -1} .

If we apply this truth to the evaluation of 6+2, we find that we instead have to evaluate:

va {6 + 1} + va {2 - 1}   (1)

and so the recursive mechanism of the processor evaluates this newly created sequence of symbols.

Now, in the process of establishing whether the last truth in V is applicable to the evaluation of this sequence the processor first performs the evaluations of va {6+1} and of va {2-1} (va operates upon the immediately following metaprimary; the notion of metaprimary is defined below);i.e. 6+1 and 2-1 are evaluated according to the truths given in V. Supposing the results of these evaluations are 7 and 1, the processor now sees whether the last truth is applicable to 7+1 and so it continues with the evaluation of 7+1.

One should notice that the result of an evaluation is not always added to V: e.g. this is not the case if we evaluate a sequence of symbols preceding the → sign or if we apply this definition of the metasymbol va (for one more example see below).

Next we describe the syntax of the metalanguage, using capital (underlined) letters as "metametasymbols".

<TERMINAL SYMBOL> IN <METAPRIMARY> CO

<METALINGUISTIC VARIABLE> IN <METAPRIMARY> CO

co IN <METAPRIMARY> IS FALSE CO

{ IN <METAPRIMARY> IS FALSE CO

} IN <METAPRIMARY> IS FALSE CO

⊀ IN <METAPRIMARY> IS FALSE CO

⊁ IN <METAPRIMARY> IS FALSE CO

<METAPRIMARY> IN <SIMPLE NAME> CO

<SIMPLE NAME> <METAPRIMARY> IN <SIMPLE NAME> CO

<SIMPLE NAME> IN <NAME> CO

<NAME> co <SIMPLE NAME> IN <NAME> CO
⦃<NAME>⦄ IN <STRING> CO

{<NAME>} IN <METAPRIMARY> CO

<STRING> IN <METAPRIMARY> CO

A TERMINAL SYMBOL is any recognizable character.

For the definition of METALINGUISTIC VARIABLE we refer to page .

Examples:

TERMINAL SYMBOL:

a
op7
αα

SIMPLE NAME:

a in <letter>

<integerlist1> is ⦃<integerlist1>⦄

{formal <identifier> co formal <identifierlist>}

NAME:

a in <letter>

0 in <digit> co 1 in <digit>

0+1 is 1 co {1+1 is 2 co 2+1 is 3} .

The following rules are part of the built-in mechanism of the processor:

1. The value of a sequence of simple names, separated by metacommas, is the sequence of the values of these simple names, separated by metacommas.

2. The value of the sequence ⦃<NAME1>⦄ is <NAME1> (i.e. the value of a string is the stripped string).

3. The value of a name, say <NAME1> , is <NAME1> , if no other information is available.

Example of rule 2:

Suppose we want to evaluate +3 and we find in V a truth of the form:

+ <unsigned integer1> is ⦃<unsigned integer1>⦄ .

We can apply this truth, since + <unsigned integer1> is an envelope of +3. Hence, the value of +3 is the value of {3} ; now, according to rule 2, {3} has the value 3 and the evaluation of +3 is finished.

If a truth has the form:
<SIMPLE NAME1> is {<NAME1> co <SIMPLE NAME 2>}
then the result of application of this truth to the evaluation of a sequence of symbols which is enveloped by <SIMPLE NAME1> is:
a. We perform the required substitutions in the right hand side.
b. The value of <NAME1> is added to V (if <NAME1> is a sequence of simple names this means that the sequence of values of these simple names, separated by metacommas, is added to V, see above).
c. We continue with the evaluation of <SIMPLE NAME2> .

During the process of establishing whether a metalinguistic variable is an envelope of a sequence of symbols, the processor has to evaluate sequences of the following form:
a in <letter> , 3+4 in <arithmetic expression> , goto L in <statement> etc. In the evaluation of these sequences, the processor uses the same method as for any other sequence, with one exception: if it establishes whether a truth is applicable to a sequence of this special, "syntactic", form, first of all the last metaprimary of the relevant part of the truth (e.g. if the truth contains the metasymbol is, then by relevant we mean the left hand side of the truth) is compared with the last metaprimary of the sequence concerned. Now, we distinguish two cases:
1. The two metaprimaries are not identical. Then the truth is not applicable to the evaluation of the sequence.
2. The two metaprimaries are identical. Then we apply the usual method to establish whether the relevant part of the truth, with the last metaprimary deleted, is an envelope of the sequence, with the last metaprimary deleted.

Example:
The truth <letter> <identifier> in <identifier> is not applicable to the evaluation of ab in <letter> , since <identifier> is not identical with <letter> .

This same truth does apply to the evaluation of ab in <identifier>, since

a. <identifier> is identical with <identifier>.

b. a in <letter> has the value true (supposing that a in <letter> is a truth in V).

c. b in <identifier> has the value true (supposing that <letter> in <identifier> is a truth in V and that b in <letter> is a truth in V).

d. in is identical with in.

Finally we mention some special features of the description of ALGOL 60, given in Chapter 3 (see also the detailed explanation in Chapter 2). We have given the list of truths in the order in which they are read in (i.e. we suppose that all the given truths are put between quotes so the effect is that we add this list to V but in such a way that truth T 1.1 (a T, followed by a number refers to the corresponding truth in the list given in Chapter 3) is the first one added to V, while T 17.69 is the last one added to V and accordingly is the first one we consult if we have to evaluate some sequence of symbols; if it is not applicable we consult 17.68 etc.). Once we have added this list to V we can ask the processor to evaluate a program. If this contains no syntactic mistakes T 1.2 will be the first one which is applicable, and so we continue by evaluating the three simple names at the right hand side of T 1.2, i.e.

1. We evaluate {βγ} which results in the addition of βγ to V.

2. We evaluate the second simple name. We try to apply the truth consisting of the two symbols βγ, then T 17.69, T 17.68,... until finally we find that we can apply T 4.9, etc.

3. We evaluate βγα: a truth which defines the value of this sequence has been added to V as a result of 2.

We remark that as soon as the evaluation of the program has started we continuously add new truths to V, and these are accordingly consulted before the "static" list T 17.69 to T 1.1 in the evaluation of the rest of the program.

We have introduced one special pair of metasymbols, i.e. $\underline{<}$ and $\underline{>}$, e.g.
$\underline{<}$type$\underline{>}$ <u>procedure</u> <id> (<idlist>); $\underline{<}$valuepart$\underline{>}\underline{<}$specpart$\underline{>}\underline{<}st\underline{>}$ <u>in</u>
<procedure declaration> (1).

If the processor wants to establish whether a truth of this form is applicable to the evaluation of a sequence of symbols, say σ , it uses a modified definition of envelope. The scheme for partitioning σ which was described on page 3 is extended in the following way: for each metavariable of the truth which is enclosed between $\underline{<}$ and $\underline{>}$ instead of < and > , we deletethe requirement that the length of the corresponding subsequence of σ be greater than zero (i.e. for those subsequences $m_i > 0$ is replaced by $m_i \geq 0$).

Example:

<u>procedure</u> P(a,b); <u>integer</u> a,b; <u>in</u> <procedure declaration>
is a sequence to which truth (1) is applicable.

(with $n=11$, $m=14$, $m_1=0$, $m_2=m_3=m_4=1$, $m_5=3$, $m_6=m_7=1$, $m_8=0$, $m_9=5$, $m_{10}=0$, $m_{11}=1$;

according to the rules for evaluation of a "syntactic" sequence we have first deleted the identical metavariables <procedure declaration> in the truth and in the sequence).

Apparently in the case of a subsequence of length zero no substitutions must be made in the rest of the truth concerned.

This extension of the metalanguage is not strictly necessary: it is possible to change our definition of ALGOL 60 in such a way that one can do without the metasymbols $\underline{<}$ and $\underline{>}$; however, at the cost of a much longer description.

Furthermore we have used some special letters as an extension of the usual alphabet (see T 17.56 to T 17.59). We assume that the programmer does not use these letters.

Moreover we introduced many new symbols by means of underlining. We suppose that these symbols have no relation to the individual letters and digits of which they are composed and that they can be recognised by the processor as independent symbols.

We use ω to denote the value of any part of a program which was left undefined or said to be undefined in [1] or which contains any offence against the rules of ALGOL 60.

3. Some examples

Example A. Greatest common divisor of two positive integers.

G 1. 1 in <digit> co
G 2. 2 in <digit> co
G 3. 3 in <digit> co
G 4. 4 in <digit> co
G 5. 5 in <digit> co
G 6. 6 in <digit> co
G 7. 7 in <digit> co
G 8. 8 in <digit> co
G 9. 9 in <digit> co
G10. <integer> <digit> in <integer> co
G11. <integer> 0 in <integer> co
G12. <digit> in <integer> co
G13. (<integer1>, <integer2>) is (<integer2>,<integer1>) co
G14. <integer1> < <integer2> →
(<integer1>, <integer2>) is
(<integer1>, va {<integer2> - <integer1>}) co
G15. (<integer1>, <integer1>) is {<integer1>} co

Remarks:

1. We have hereby defined the usual Euclidean Algorithm with repeated subtraction instead of division.
2. The list of truths given above should be extended with truths defining the value of <integer> < <integer> and <integer> - <integer> . For these definitions we refer to the corresponding truths given in our description of ALGOL 60.
3. It is easy to extend this definition to the g.c.d. of two arbitrary integers.
4. See also example E.
5. We have given the list of truths in the order in which they are

read in; accordingly, G 15 is the first truth consulted, the next one is G 14 etc.

We give an example (in an obvious notation):

$$(42,105) \overset{G14}{\Rightarrow} (42,\underline{va}\ \{105-42\}) \Rightarrow (42,63) \overset{G14}{\Rightarrow} (42,\underline{va}\{63-42\}) \Rightarrow$$
$$(42,21) \overset{G13}{\Rightarrow} (21,42) \overset{G14}{\Rightarrow} (21,\underline{va}\{42-21\}) \Rightarrow (21,21) \overset{G15}{\Rightarrow} \{21\} \Rightarrow 21.$$

Example B. Lexicographical ordering.

L 1. a in <letter> co

L 2. b in <letter> co

L 3. c in <letter> co

L 4. d in <letter> co

L 5. e in <letter> co

L 6. <word> <letter> in <word> co

L 7. <letter> in <word> co

L 8. <word> ≺ <word> is false co

L 9. <letter1> ≺ <letter2> →
<letter1> <word>≺<letter2><word> co

L10. <letter1>≺<letter2> →
<letter1><word>≺<letter2> co

L11. <letter1>≺<letter2> →
<letter1>≺<letter2><word> co

L12. <letter1><word1>≺<letter1><word2> is
<word1>≺<word2> co

L13. <letter1>≺<letter1><word> co

L14. <letter1><word>≺<letter1> is false co

L15. <letter2>≺<letter3>→ <letter1>≺<letter3> is
<letter1>≺<letter2> co

L16. <letter>≺ a is false co

L17. a ≺ b co

L18. b ≺ c co

L19. c ≺ d co

L20. d ≺ e co

L21. <word1>≺<word1> co

Suppose one wants to evaluate dbc≺dee.
According to the first applicable truth, L 12, the value of this sequence of symbols is equal to the value of bc≺ee.(1)
L 21 to L 10 are not applicable to (1) so now we try L 9. According to the definition of the metasymbol → we have to evaluate b≺e (i.e. the sequence at the left hand side of → , after the required substitutions) in order to establish whether L 9 is applicable.
We try to apply L 15 to the evaluation of b≺e; this leads to the evaluation of the left hand side: <letter2>≺e. (2)
This sequence contains a metavariable and we now have to apply the special method for evaluation of (2).
We find that (2) is an envelope of L 20.
Hence the value of (2) is true, we may apply L 15, we substitute d for <letter2> and we evaluate b≺d instead of b≺e.
By exactly the same process we find that the value of b≺d is equal to the value of b≺c but according to L 18 this has the value true.
Now we know that L 9 is applicable to the evaluation of bc≺ee and the result of application is that bc≺ee has the value true.
Thus, our final result is that we find true as the value of dbc≺dee.

Example C. Definition of the language of Turing machines.

T 1. <state1> <symbol1> <symbol2> <state2> →
<tape1> <state1> <symbol1> <tape2> is
<tape1> <state2> <symbol2> <tape2> co

T 2. <state1> <symbol1> L<state2> →
<tape1> <symbol2> <state1> <symbol1> <tape2> is
<tape1> <state2> <symbol2> <symbol1> <tape2> co

T 3. <state1> <symbol1> L <state2> →
<state1> <symbol1> <tape1> is
<state2> 0 <symbol1> <tape1> co

T 4. <state1> <symbol1> R <state2> →
<tape1> <state1> <symbol1> <tape2> is
<tape1> <symbol1> <state2> <tape2> co

T 5. <state1> <symbol1> R <state2> →
<tape1> <state1> <symbol1> is
<tape1> <symbol1> <state2> 0 co

T 6. 0 in <symbol> co

T 7. <symbol> in <tape> co

T 8. <tape> <symbol> in <tape> co

A possible "program" is:

T 9. 1 in <symbol> co

T10.	q <state>	in	<state>		co
T11.	q	in	<state>		co
T12.	q	1	0	q	co
T13.	q	0	R	qq	co
T14.	qq	1	R	qq	co
T15.	qq	0	R	qqq	co
T16.	qqq	1	0	qqq	co

Remarks:

1. This definition is a transcription of the definitions in [2] , Chapter 1.

T 1	corresponds to [2] , Ch.1, def.1.7	(1)
T 2	" " " " " "	(4)
T 3	" " " " " "	(5)
T 4	" " " " " "	(2)
T 5	" " " " " "	(3).

2. We assume that T 1 to T 8 are in V, whenever one wants to evaluate a program written in the language of Turing machines. Truths T 9 to T 16 may be replaced by some other truths, i.e.
   a) one may want to extend the alphabet by adding e.g.
      ε in <symbol> or
      η in <symbol> etc.
   b) one may want to change the syntactic definition of <state>.
   c) T 12 to T 16 (cf. [2] Ch.1, example 3.1) which form a program for addition may be replaced by another list of "quadruples".

3. As usual we assume that T 1 to T 16 are put between quotes, after which they are read in by the processor.

If one asks the machine to evaluate e.g. q1110110 the result of application of T 1 to T 16 is that this value is 0110qqq010.

Example D. Definition of some finite automata.

The following examples are based on: M.O. Rabin and D. Scott: Finite Automata and their Decision Problems [4] .

D 1. One way, one tape, deterministic ([4] , definitions 1,2)

```
RS 1. <tape> <symbol> in <tape> co
RS 2. <symbol> in <tape> co
RS 3. <state> <symbol> is {false} co
RS 4. <state1> <symbol1> <state2> →
      <state1> <symbol1> <tape1> is
               <state2> <tape1> co
RS 5. <state1> <symbol1> <final state> →
      <state1> <symbol1> co
```

A possible "program" is:

```
RS 6.  a  in <symbol>  co
RS 7.  b  in <symbol>  co
RS 8.  1  in <state>   co
RS 9.  2  in <state>   co
RS10.  3  in <state>   co
RS11.  3  in <final state>  co
RS12.  1 a 2           co
RS13.  2 a 3           co
RS14.  3 a 2           co
RS15.  1 b 1           co
RS16.  2 b 2           co
RS17.  3 b 1           co
```

Again we assume that RS 1 to RS 5 are always in V, whereas the "programmer" may replace RS 6 to RS 17 by some other truths. Once we have read in RS 1 to RS 17 we may ask the processor to evaluate e.g. 1 aba (which is found to have the value true, i.e. the tape aba is accepted by the automaton if its initial state is 1) or 1 bab which has the value false.

D 2. Two way, one tape, deterministic ([4], definitions 13,14)

```
RSS 1.  <tape>  <symbol>  in  <tape>  co
RSS 2.          <symbol>  in  <tape>  co
RSS 3.  ≤tape≥  <state> <symbol>  ≤tape≥  is  {false}  co
RSS 4.  <state1> <symbol1> L <state2> →
        ≤tape1≥ <symbol2> <state1> <symbol1> ≤tape2≥  is
        ≤tape1≥ <state2> <symbol2> <symbol1> ≤tape2≥  co
RSS 5.  <state1> <symbol1> R <state2> →
        ≤tape1≥  <state1> <symbol1> <tape2>  co
        ≤tape1≥  <symbol1><state2>  <tape2>  co
RSS 6.  <state1> <symbol1> C <state2> →
        ≤tape1≥ <state1> <symbol1> ≤tape2≥  is
        ≤tape1≥ <state2> <symbol1> ≤tape2≥  co
RSS 7.  <state1> <symbol1> <final state> →
        ≤tape≥  <state1> <symbol1>          co
```

A possible "program" is:

```
RSS 8.     a  in <symbol>   co
RSS 9.     b  in <symbol>   co
RSS10.     1  in <state>    co
RSS11.     2  in <state>    co
RSS12.     3  in <state>    co
RSS13.     3  in  <final state> co
RSS14.     1 a R 2          co
RSS15.     2 a L 1          co
RSS16.     3 a R 3          co
RSS17.     1 b R 3          co
RSS18.     2 b C 3          co
RSS19.     3 b R 2          co
```

After reading in RSS 1 to RSS 19 one may ask the processor to evaluate e.g. 1 aba or 1 bab.

D 3. One way, one tape, non deterministic ([4], definitions 9,10)

RRS 1. <tape> <symbol> in <tape> co

RRS 2. <symbol> in <tape> co

RRS 3. <state list> <state> in <state list> co

RRS 4. <state> in <state list> co

RRS 5. <state> <tape> is false co

RRS 6. <state1> <symbol1> <state list1> →
<state1> <symbol1> <tape1> is
<state list1> <tape1> co

RRS 7. <state1> <symbol1> <state list> <final state> <state list> →
<state1> <symbol1> is
{{<state list> <tape>} co {true}} co

RRS 8. <state1> <state list1> <tape1> is
{<state1> <tape1> co <state list1> <tape1>} co

Remarks:

1. We assume that a state list is uniquely deconcatenable into its constituent elements.
2. Suppose we are evaluating 1 aba and we find 1 a 2 3 in V. Then we continue with the evaluation of 2 ba and of 3 ba (RRS 6, RRS 8). If after the next step we find that the value of 2 ba is equal to the value of 4 a, say, while in V we find 4 a 5, where 5 is final, then the result of application of RRS 7 is that we add to V:

   <state list> <tape> (1)

   and: true.

   The effect of (1) is that all the remaining sequences of symbols which the processor still has to evaluate, e.g. 3 ba, have the value true.

An equivalent definition of a one way, one tape, non deterministic automaton is the following:

RRS 9. <state> <state list> in <state list> co

RRS10. <state> in <state list> co

RRS11. <state> <symbol> in <mixed tape> co

RRS12. <mixed tape> <mixed tape> in <mixed tape> co

RRS 13. <mixed tape1> <state1> <symbol1> <state2> <tape1> b
is <mixed tape1> <state1> <symbol1> <tape1> b co

RRS 14. <mixed tape1> <state1> <symbol1> is
<mixed tape1> <state1> <symbol1> b co

RRS 15. <state> <tape> b is {false} co

RRS 16. <state1> <symbol1> <state list> <state2> <state3>
<state list> →
<mixed tape1> <state1> <symbol1> <state2> <tape1> b
is <mixed tape1> <state1><symbol1><state3><tape1> co

RRS 17. <state1><symbol1><state2><state list1> →
<mixed tape1> <state1><symbol1><tape1> is
<mixed tape1> <state1><symbol1><state2><tape1> co

RRS 18. <state1><symbol1><state list><final state><state list>
→ <mixed tape><state1><symbol1> co

We demonstrate these truths by a sample program:

RRS 19. 1 in <state> co
RRS 20. 2 in <state> co
RRS 21. 3 in <state> co
RRS 22. 3 in <final state> co
RRS 23. a in <symbol> co
RRS 24. b in <symbol> co
RRS 25. 1 a 2 1 co
RRS 26. 2 a 3 co
RRS 27. 3 a 2 3 co
RRS 28. 1 b 1 2 co
RRS 29. 2 b 2 co
RRS 30. 3 b 1 co

Now, by applying these truths together with RRS 9 to RRS 18, we find that 1 bab has the value false with the following intermediate results:

1 bab $\overset{RRS17}{\Rightarrow}$ 1 b 1 ab $\overset{RRS17}{\Rightarrow}$ 1 b 1 a 2 b $\overset{RRS14}{\Rightarrow}$ 1 b 1 a 2 b b $\overset{RRS16}{\Rightarrow}$ 1 b 1 a 1 b $\overset{RRS14}{\Rightarrow}$ 1 b 1 a 1 b b $\overset{RRS13}{\Rightarrow}$ 1 b 1 a b b $\overset{RRS16}{\Rightarrow}$ 1 b 2 a b $\overset{RRS17}{\Rightarrow}$ 1 b 2 a 3 b $\overset{RRS14}{\Rightarrow}$ 1 b 2 a 3 b b $\overset{RRS13}{\Rightarrow}$ 1 b 2 a b $\overset{RRS13}{\Rightarrow}$ 1 b a b b $\overset{RRS15}{\Rightarrow}$ {false} ⇒ false.

One might think of this process in terms of a search along a tree, where b indicates a search from bottom to top.

D 4. One way, two tape, deterministic ([4], definitions 15,16)

RRSS 1. <tape> <symbol> in <tape> co

RRSS 2. <symbol> in <tape> co

RRSS 3. F in <FS> co

RRSS 4. S in <FS> co

RRSS 5. (<tape>,<state> <symbol>) is {false} co

RRSS 6. (<state><symbol> , <tape>) is {false} co

RRSS 7. <state1> <symbol1> S <state2> →
(<tape1>, <state1> <symbol1> <tape2>) is
(<tape1>, <state2> <tape2>) co

RRSS 8. <state1><symbol1> F <state2> →
(<tape1>, <state1><symbol1><tape2>) is
(<state2> <tape1>, <tape2>) co

RRSS 9. <state1><symbol1> S <state2> →
(<state1><symbol1><tape1>, <tape2>) is
(<tape1>, <state2><tape2>) co

RRSS10. <state1><symbol1> F <state2> →
(<state1><symbol1><tape1> , <tape2>) is
(<state2> <tape1>, <tape2>) co

RRSS11. <state1><symbol1><FS> <final state> →
(<state1><symbol1>, <tape>) co

RRSS12. <state1> <symbol1><FS><final state> →
(<tape>, <state1> <symbol1>) co

Remark:

F refers to the first tape, S to the second one, e.g. if the reading head reads on tape 1 and if an S occurs in the "instruction" concerned, control is changed to the second tape.

Example E. The algorithm for the greatest common divisor of two natural numbers in the normal form of A.A. Markov ([3], page 105).

M 1.	1 **in** <symbol> **co**
M 2.	* **in** <symbol> **co**
M 3.	a **in** <symbol> **co**
M 4.	b **in** <symbol> **co**
M 5.	c **in** <symbol> **co**
M 6.	<symbol> **in** <tape> **co**
M 7.	<tape><symbol> **in** <tape> **co**
M 8.	≤tape1≥*≤tape1≥ **is** ≤tape1≥ ≤tape2≥ **co**
M 9.	≤tape1≥ c≤tape2≥ **is** ≤tape1≥1 ≤tape2≥ **co**
M10.	≤tape1≥ a≤tape2≥ **is** ≤tape1≥c ≤tape2≥ **co**
M11.	≤tape1≥ b≤tape2≥ **is** ≤tape1≥1 ≤tape2≥ **co**
M12.	≤tape1≥1*≤tape2≥ **is** ≤tape1≥*b≤tape2≥ **co**
M13.	≤tape1≥1*1≤tape2≥ **is** ≤tape1≥a*≤tape2≥ **co**
M14.	≤tape1≥1a≤tape2≥ **is** ≤tape1≥a1≤tape2≥ **co**

Remarks:

1. If one denotes the natural number N by a sequence of N symbols 1 then this list of truths defines the g.c.d. of a pair of natural numbers, say N and M, represented by concatenation of:
   a sequence of N symbols 1,
   the symbol * ,
   a sequence of M symbols 1.
2. It seems probable that every algorithm in Markovs normal form can be rewritten as an equivalent sequence of truths in a manner similar to that in this example.

References.

1. J.W. Backus et al. Revised Report on the Algorithmic Language ALGOL 60, edited by P. Naur. Regnecentralen, Copenhagen, 1962.

2. M. Davis Computability & Unsolvability. McGraw-Hill, New York, 1958.

3. A.A. Markov Theory of Algorithms. Moscow, Leningrad, 1954. (Published for the N.S.F. and the Department of Commerce, U.S.A., by the Israel Program for Scientific Translations, 1961).

4. M.O. Rabin and D. Scott Finite Automata and their Decision Problems. I.B.M. J. of Research and Development, 3, 1959, pp. 114-125.

5. A. van Wijngaarden Generalized ALGOL. Symbolic Languages in Data Processing. Proc. Symposium, Rome, 1962, pp. 409-419.

6. A. van Wijngaarden Recursive definition of syntax and semantics. IFIP Working Conference on Formal Language Description Languages, Vienna, 1964 (to be published).

Chapter 2

1. Introduction

In the sequel an explanation will be given of the techniques, used in Chapter 3, for the description of almost all of ALGOL 60. First of all we give a list of subjects which are not treated (i.e. 1,2,3,4) or which are treated incorrectly (i.e. 5,6).

1. real arithmetic.
   In ALGOL 60 no exact arithmetic has been specified ([1], 3.3.6), this specification belongs to the accompanying information which should be given by the programmer (cf. also [1], 1, footnote 1). Thus, whenever one wants to execute a program in which real arithmetic is used one has to extend our list of truths in Chapter 3 with additional truths specifying the arithmetic one wants to use. Moreover, the declarator <u>real</u> should be introduced and one should give the description of its consequences for declarations, assignment statements etc.
2. comment conventions and) <letterstring> :(as parameter delimiter. It is of course easy to include those features and they were only left out in order not to increase the length of the description.
3. procedure bodies in code and strings as actual parameters.
4. no attention was paid to the inclusion of standard functions.
5. expressions containing formal parameters may become undefined after name replacement as in the following example:
   <u>if</u> boolean <u>then</u> f <u>else</u> g might be replaced by
   <u>if</u> <u>true</u> <u>then</u> 3 else $a \vee b$.

   The mechanism of the description gives 3 as the value of the last expression whereas it should be ω.
6. A conditional statement of the form <u>if</u> <bexp1> <u>then</u> <unconditional statement1> , where <bexp1> has the value <u>false</u> is equivalent to the dummy statement only if the evaluation of

<bexp1> has no side effects. Mutatis mutandis this holds for a goto statement leading to an undefined switch designator.

Furthermore in the next two cases one might prefer another interpretation.

1. Only the static definition of own is given.
2. Specifications of non value parameters are ignored.

In general, whenever in a program something occurs which was left undefined in the report or said to be undefined or forbidden, the value of the program is ω. However, sometimes we could not avoid a choice, e.g. regarding the order of evaluation of the value parameters where we chose the order given in the formal parameter list. Furthermore primaries in the expressions are evaluated in order from left to right.

Next we shall explain the structure of the description.
The three main difficulties proved to be:

1. The concept of locality.
2. Labels and goto statements.
3. The requirement that all identifiers of a program be declared, even in parts of the program which are not executed, e.g. <u>begin</u> <u>if</u> <u>false</u> <u>then</u> i:=0 <u>end</u> is not a correct ALGOL program.

The first point made it necessary to introduce the notion of block number (see below); the last two required the equivalent of a prescan. In the evaluation of a program we distinguish the following stages:

1. Somewhere in V is a language rule like <program1> <u>is</u> ....., (see T 1.2) and the syntactic definition of a program is also given (see T 2.28, T 2.29 etc.).
   Thus, if there is a syntactic mistake in the program, truth T 1.2 will not be applicable and the only one which does apply is truth T 1.1: <name> <u>is</u> $\nmid\omega\nmid$.
   Hence, the syntactic correctness of the program is checked automatically by the mechanism of the metalanguage.
2. In the first stage of the prescan we note the different names which are introduced either by explicit declaration or by standing as a

label or in a formal parameter list. In the second stage we check whether every statement of the program (including statements in procedure declarations) contains only declared identifiers. Both of these operations are defined recursively for each block. In both phases we use a static block number.

3. In the first stage of the execution we scan the program for the occurrence of labels which we then supply with the dynamic block number of the smallest embracing block in order to make it possible to restore the correct block number if a goto statement leaves a block.
   Finally the program is executed.

2. The determination of the block number (see T 1.17, T 1.18, T 1.19, T 1.20, T 1.21, T 10.16 and T 10.17).

In the sequel we shall use the abbreviation bn for block number.
According to the syntax a bn consists of a sequence of β's, followed by a γ, possibly followed by a bn, possibly followed by a δ followed by a bn.
At the beginning of the evaluation of a program the bn is set to βγ; i.e. βγ is added as a truth to V (see T 1.2).
Suppose at a given moment the bn is <βγs1>≤δβγs1≥ and we are entering a new block. Now there are two possibilities. Either somewhere in V we find <βγs1><βγ1>≤δβγs≥, meaning that the new block is parallel to an earlier one, possibly itself during execution state, in which case we add <βγs1>β<βγ1>≤δβγs1≥ to V; or else we find no such thing, in which case we write in V: <βγs1>βγ≤δβγs1≥. If the current bn is <βγs1><βγ>≤δβγs1≥(by current we mean that this is the last entry in V of the syntactic form of a bn) upon exit from a block, we add <βγs1>≤δβγs1≥ to V. The value of the last <u>end</u> of the program is defined in T 1.21 and is explained below.
The rules governing block entrance and exit hold both for the prescan and execution phase of "normal" blocks and procedure bodies. If a procedure is declared in a block with bn <βγs1>≤δβγs≥ and called from a block with bn <βγs2>≤δβγs1≥ , we add to V: <βγs1>δ<βγs2>≤δβγs1≥, upon

entrance to the procedure. After this we perform the entrance to the procedure body itself (which is always made into a block) according to the rules for block entrance given above.
Upon exit from a procedure, we search for the current bn. If this has the form ≤βγs >δ<βγs2>≤δβγs1≥ we add <βγs2>≤δβγs1≥ to V. By means of the last two rules, which of course only hold in the execution phase, we restore the correct bn after exit from a procedure.

In this way, at every moment the last item in V which has the syntactic form of a bn defines the current bn. We use this whenever we process an identifier; in all possible cases the identifier is first extended with the current bn.

3. The prescan. (Truths T 1.2 to T 1.5, T 3.1 to T 3.12, T 4.9, T 4.13, T 5.15, T 5.28, T 6.6, T 7.29, T 7.30, T 11.8 to T 11.15).

Clearly we need a technique to read a program and process the declarations occurring in it (because we want to check whether all identifiers are declared) but without executing it. This was accomplished by introducing the concept of an artificial label, consisting of the bn of the block which we are scanning, followed by one or more α's; by labelling successively each declaration and statement of the block and by adding to V a rule for evaluating each of those labels.
Some examples will perhaps clarify this:
(actually, in the examples we use an unimportant simplification so they do not exactly correspond with the rules given in our list).
Example A. At a given moment during the prescan we want to determine the value of:

<βγs1><αs1>:<type1><id1>;≤decl list1≥ ≤st list1≥ <u>end</u>

(with an obvious definition of <decl list> and <st list> , see T 2.6 and T 2.20). According to truth T 4.9 this value is given by:

1. <type1> <id1> <βγs1> is added to V, provided that the same identifier had not been declared before in the same block, in which case ω is added to V. Now we know in the second stage of the prescan that <id1> has been declared and is of type <type1> in

the block with bn <βγs1>.

2. We add to V: <βγs1><αs1> is {ǂ<βγs1><αs1> is {ǂτ<βγs1><αs1> is {<type1><id1> co τ<βγs1><αs1>α}ǂ co <βγs1><αs1>α}ǂ co<βγs1><αs1>α}. (1)

When, later on, we ask for the value of <βγs1><αs1> (in phase 2 of the prescan), the result of the application of (1) is that we just add

<βγs1><αs1> is {ǂτ<βγs1><αs1> is {<type1><id1> co τ<βγs1><αs1>α}ǂco <βγs1><αs1>α} (2)

to V and continue by evaluating <βγs1><αs1>α .

When, in phase 1 of the execution, we ask for the value of <βγs1><αs1>, the result of application of (2) is that we only add to V a new truth, i.e.

τ<βγs1><αs1> is {<type1><id1> co τ<βγs1><αs1>α} , (3)

and again ask for the value of <βγs1><αs1>α . (For the meaning of the extra symbol τ see below.)

When, in phase 2 of the execution, we ask for the value of τ<βγs1><αs1>, by application of (3) we find that we have to do two things:

a. determine the value of <type1><id1>;according to truth T 4.10 we now add to V: <type1><id1> followed by the current dynamic bn.

b. continue by evaluating τ<βγs1><αs1>α.

3. We proceed to determine the value of

<βγs1><αs1>α : <decl list1><st list1> end.

Example B. Suppose we want to know the value of

<βγs1><αs1> : <ass st1> <special st list1> end

(see truth T 2.21 for definition of <special st list>).

According to truth T 3.9 this is given by:

1. We add to V: <βγs1><αs1> is {<ass st1> in <decl ass st> co ǂ<βγs1><αs1> is {ǂτ<βγs1><αs1> is {<ass st1> co τ<βγs1><αs1>α}ǂ co <βγs1><αs1>α}ǂ co <βγs1><αs1>α} .

Thus, in phase 1 of the prescan, we just write down this rule in V. In phase 2 of the prescan, we ask whether <ass st1> contains only declared identifiers (see also below), we add to V a new rule for evaluating <βγs1><αs1> , and we continue by evaluating <βγs1><αs1>α . In phase 1 of the execution, we add to V a rule, giving the value of

τ<βγs1><αs1>, and determine the value of <βγs1><αs1>α.
In phase 2 of the execution, we have to perform <ass st1> and continue with τ <βγs1><αs1>α .
2. We proceed to determine the value of

<βγs1><αs1>α : <special st list1> end.

Example C. The value of

<βγs1><αs1> : <label1> : <st list1> end

is given, according to truth T 3.12 by:

1. label <label1><βγs1> is added to V (with the same precautions as in example A). Hence, we know in the second phase of the prescan that <label1> is a label in the block with bn <βγs1>.
2. We add to V: <βγs1><αs1> is {⦃<βγs1><αs1> is
{<label1> op3 <βγs1><αs1> co ⦃τ<βγs1><αs1> is τ<βγs1><αs1>α⦄co
<βγs1><αs1>α}⦄ co <βγs1><αs1>α} .

In phase 2 of the prescan, we write down a new definition of <βγs1> <αs1> and continue with the evaluation of <βγs1> <αs1>α . In phase 1 of the execution we have to do three things:
a. We perform op3 (see T 3.15) upon <label1> with two effects:
a1. <label1> is supplied with the current dynamic bn.
a2. <label1> is supplied with the artificial label τ<βγs1><αs1>α , so that when we execute a goto statement leading to <label1> we know the right way to continue the program since we have only to ask for the value of τ<βγs1> <αs1>α ,
b. we write down the definition of τ<βγs1><αs1>.
c. we evaluate <βγs1><αs1>α.

In phase 2 of the execution, if we ask for the value of τ<βγs1><αs1> , it appears that we have to go on with the evaluation of τ<βγs1><αs1>α.
3. We determine the value of <βγs1> <αs1>α: <st list1> end.

Example D. The value of <βγs1> <αs1> : end is given, according to truth T 1.5 by:
1. We add to V: <βγs1> <αs1> is {end co ⦃ <βγs1><αs1> is
{⦃τ<βγs1> <αs1> is end⦄ co τ<βγs1>α}⦄} .

This has the following meaning:

In phase 2 of the prescan, we determine the value of end (see T 1.22) and write down a new definition of <βγs1><αs1>.

In phase 1 of the execution, we give a definition of τ<βγs1><αs1> and ask for the value of τ<βγs1>α , since, when part 1 of the execution phase is finished, we can really execute the block and so now we ask for the value of the first artificial label of that block. The reason for the addition of an extra symbol τ now becomes clear. If we had not used this device then during a second activation of a block (for example if we return by means of a goto statement), we would not have been able to distinguish between stage 1 of the execution of a block and stage 2. If, in phase 2 of execution of the block, we ask for the value of τ<βγs1><αs1> we see that we have to determine the value of end. Then the execution of this block is finished and we can continue with the next statement.

2. We ask for the value of <βγs1>α since the first stage of the prescan is now finished and we return to the beginning of the block in order to start with the second phase of the prescan by asking for the value of the first artificial label of that block.

Example E. The mechanism for recursively calling the prescan at every static block entrance is described in truths T 3.10, T 1.3, T 1.4. According to T 3.10, when, during the prescan, we process a block we have to do the following:

Suppose we want to determine the value of

<βγs1><αs1> : <block1> <special st list1> end.

It it given by:

1. We define <βγs1><αs1> in such a way that in the second phase of the prescan we must evaluate:

a. <block1> in <decl block> , see below.

b. we determine the first artificial label of this block and store it in V.

c. we define a new rule for evaluating <βγs1><αs1>.

d. we continue by evaluating <βγs1><αs1>α .

2. In the first stage of the execution we give a new definition of <βγs1><αs1> and evaluate <βγs1><αs1>α.

3. In the second phase of the execution we ask for the value of the first artificial label of this block (which we can find because of 1.b) and if the execution of this block is finished, we next evaluate τ < βγs1><αs1>α which in general will give rise to the evaluation of the statement following the block. Here, it is clear that a difficulty arises: it is perfectly possible that during execution of the block we jump out of it to some non local label. And so, if we have arrived at the last end of the program the recursive mechanism of the processor still wants to evaluate τ<βγs1><αs1>α , whereas the execution has actually been completed. Now to solve this problem we define as the value of the last (dynamic) end: {̸<name>}̸ (see T 1.21).
Thus, the effect is that <name> appears as a truth in V and everything which the processor evaluates hereafter, for example τ<βγs1><αs1>α , has the value true, which is completely harmless.

4. In the first stage of the prescan we evaluate

<βγs1><αs1>α : ≤special st list1≥ end.

According to T 1.3 the value of

begin <decl list1> ≤st list1≥ end in <decl block>

is given by:

1. Determine the value of begin (T 1.17, T 1.18, T 1.19).

2. Determine the value of ≤decl list1≥ ≤st list1≥ end.

According to truth T 1.4 the value of

<decl list 1> ≤st list1≥ end

is given by:

(suppose the current bn is <βγs1>)

1. <βγs1>α is {{̸ <βγs1>α is { begin co {̸ τ<βγs1>α is τ<βγs1>αα}̸ co <βγs1>αα}}̸ co <βγs1>αα} ,

which has the following meaning:

a. in the second phase of the prescan we define a new value of <βγs1>α and evaluate <βγs1>αα.

b. in the first phase of the execution we evaluate begin, define the value of τ<βγs1>α , and evaluate <βγs1>αα.

c. in the second phase of the execution we have to evaluate τ<βγs1>αα.
2. We continue to determine the value of

<βγs1>αα : <decl list1> <st list1> end

and so we have recursively initialized the prescan of this new block.

Next we give a summary of the treatment in the prescan rules of the other declarations and statements:

1. array, switch, and procedure declarations: analogously to type declarations; see also the separate treatment of these cases.
2. conditional statements are reduced to one special form:
   if <bexp> then goto <dexp> .
3. for statements: see section 10.
4. procedure statements and goto statements: analogously to assignment statements.
5. compound statement: we strike out begin and end.

If, finally, the prescan is finished (i.e., if the evaluation of the second sequence of symbols in the right hand side of truth T 1.2 is finished), we ask for the value of βγα, which is the first artificial label of the program and so we start execution phase 1.

4. The requirement that all identifiers of a program be declared.

In phase 2 of the prescan we ask for the value of expressions of the following form: <ass st1> in <decl ass st> , <dexp1> in <decl dexp> etc.
Evaluating of these expressions is possible because in many places in our list of truths we included among the syntactic definitions besides the ones in [1] also truths of the form:

<decl int var> in <decl primary> ,
<decl factor> in <decl term>,
<decl saexp> <relop><decl saexp> in <decl b primary> ,
<decl int var> in <decl int left part> ,
<decl int left part list> <decl aexp> in <decl ass st> etc.

This needs explanation in two respects:

1. Where possible, we also check whether the expression on the right hand side of an assignment statement is of the same type as the entries in the left part list. This is achieved by introducing the notions of integer variable, which is either an integer variable identifier or something of the form <int array id> [<subexplist>], of boolean variable etc.

Once we know this it is of course easy to define integer left part list etc.

2. Suppose the current bn is <βγs1> and we want to know whether <id1> is a declared integer variable identifier. First we replace this question by: <id1> <βγs1> in <decl int var id> . (1)

Then we scan V for the occurrence of integer <id1> <βγs1>.

If we have success we know that the value of (1) is true.

If not, we look for something of the form formal <id1> <βγs1> (this might have resulted from the treatment of a formal parameter list during the prescan, see T 7.32). If we succeed we also define the value of (1) to be true. Otherwise, if we want to determine the value of <id1><βγs1><βγ> in <decl int var id>, we ask for the value of <id1> <βγs1> in <decl int var id> .

Apparently, we now search the smallest embracing block for an occurrence of either integer <id1> <βγs1> or formal <id1><βγs1> .

If we have no success we again search an embracing block until we finally ask for the value of <id1>βγ in <decl int var id>, and now if we fail again it is clear that <id1> has not been declared as an integer and so the value of <id1>βγ in <decl int var id> is false.

Remarks:

1. We also include a check whether the number of parameters in a procedure statement equals the number given in the corresponding declaration (see e.g. T 7.26).
2. When a statement turns out to contain an identifier which has not been declared we add ω to V since in this case the only truth which applies is T 1.1.

5. Type declarations and the evaluation of a simple variable (T 4.1 to T 4.28)

Essentially the result of processing type declarations of non own simple variables is simply that the declared identifier supplied with the current bn and the given type is added to V (during the prescan we check whether this identifier has not been declared already in the same block. If this is the case ω is added to V).

If a simple variable is declared own, we have to do somewhat more:

1. Normally with non own type declarations the result of the prescan is that in phase 2 of the execution we have to evaluate τ<βγs1><αs1> by applying a rule like:

   τ<βγs1><αs1> is {<type1> <id1> co τ<βγs1><αs1>α} .

   With own variables this is replaced by:

   τ<βγs1><αs1> is {<βγs1><αs1> : own <type1><id1> co τ<βγs1><αs1>α}. (1)

2. The first time we enter the block during execution phase 2 the result of applying (1) is the following:

a. <id1> is supplied with the current bn, say <βγs2> , and with the type <type1> and is added to V. Hence, this is just the same as with non own simple variables.

b. To V is added:

   τ<βγs1><αs1> is {<βγs1><αs1> : own <type1><id1><βγs2> co τ<βγs1><αs1>α} .

   Thus, we remember the bn of the block <βγs2> in which we first met own <type1><id1>.

3. The next time we have to evaluate τ<βγs1><αs1> we find that we have to apply a rule of the following form:

   τ<βγs1><αs1> is {<βγs1><αs1>: own <type1><id1><βγs2> co τ<βγs1><αs1>α},

   which has been added to V as a result of 2.b and which has the following effect:

a. <id1> is supplied with the current bn, say <βγs3> and with the type <type1> and is added to V.

b. To V is added <id1> <βγs3> is <id1><βγs2> , so that if we want to know the value of <id1> at the moment that <βγs3> is the current bn and if moreover there has been no assignment to <id1> during this activation of the block we find that we have to search for a value of <id1> which may possibly have been assigned to it in the previous activation of the same block which activation had as bn <βγs2> . (In order to understand the foregoing completely one also has to know how the evaluation of an assignment statement is defined.)

c. To V is added:
τ<βγs1><αs1> is{<βγs1><αs1>: own <type1><id1><βγs3> co τ<βγs1><αs1>α}.
so that if we return the next time to the same block we have remembered the bn of this activation, i.e. <βγs3>.

If we ask for the value of a simple variable, say <id1> , and if the current bn is <βγs1> we continue by determining the value of <id1><βγs1> (see T 4.16).
If <id1> had been declared in this block (so if in V we find <type><id1><βγs1>), and if we arrive at truth T 4.21 then clearly no assignment to <id1> has taken place (otherwise we would first have met a truth of the form <id1><βγs1> is <in> or <id1><βγs1> is <logical value> as the result of such as assignment, see also definition of assignment statements). Hence, in this case we find that the value of <id1><βγs1> is ω. Another possibility is that <id1><βγs1> is a formal parameter which is called by name and which has an expression <exp1> as its corresponding actual parameter (see truth T 4.19; in order to understand this one must also know how the evaluation of a procedure statement is defined).
Then we do the following:

1. The current bn is preserved.
2. The bn of the block in which the procedure statement occurs is added to V.
3. <exp1> is evaluated and a rule which contains this result, i.e. result <id1><βγs1> , is added to V.
4. The bn which was preserved in 1. is restored.
5. And now the value of <id1><βγs1> is the value of result <id1><βγs1>.

When neither truth T 4.19 nor T 4.21 applies and if we assume that <id1> is not a function designator (this case is treated below), then the effect of asking for the value of <id1><βγs2><βγ>is that we determine the value of <id1><βγs2> i.e. we search the smallest embracing block (here we suppose that <βγs2><βγ> is identical with <βγs1>). One should notice that here and in similar cases where we search an embracing block the current bn is not changed. We again try to apply the rules T 4.19 and T 4.21, in case of failure try the embracing block etc. until after repeated failure we ask for the value of <id1>βγ which is obviously ω because now there no longer is an embracing block.

6. Array declarations and the evaluation of a subscripted variable (T 5.1 to T 5.44)

The prescan rule for the treatment of an array declaration is analogous to the one for type declarations (see T 5.15 and T 4.9). However, in the first phase of the prescan, we have to do two things:

1. Just as in the case of type declarations we check whether the array identifier has not been declared before in this block and, if not, we add to V a note describing the nature of the identifier and the block in which it was declared.
2. We check whether the expressions occurring in the bound pair list contain only declared identifiers. In view of [1], 5.2.4.2 we first activate the bn of the smallest embracing block, then we evaluate <bplist1> in <decl bplist> , and finally we restore the old bn.

As might be expected, the rules for evaluating an array declaration during the execution phase are somewhat more complicated:

1. We define syntactically an integer bound pair list as a bound pair list which contains only integers.
2. If the array segment has a bound pair list which is not an integer bound pair list we first evaluate every expression in the bound pair list, again after first activating the smallest embracing block and later on reactivating the bn of the current block.

3. A case like ... <u>integer</u> <u>array</u> a,b[1:10]... is transformed into ... <u>integer</u> <u>array</u> a [1:10] ; <u>integer</u> <u>array</u> b[1:10] ...
4. Finally we add to V a note of the following form:
<type> <u>array</u> <id1><βγs1>[<int bplist>]
where <βγs1> is the current dynamic bn. This last entry will be used later on if we want to perform an assignment to a subscripted variable.

The treatment of own arrays follows easily from a combination of the definition of own type declarations and of the definition given above of non own arrays. Only one extra difficulty arises: According to [1], 5.2.5 when we ask for the value of a subscripted variable which corresponds to an own array and which has obtained a value in a former activation of the block we have to test whether the subscripts are within the most recently calculated subscript bounds. This is accomplished by the last part of the right hand side of T 5.33:
Only if the subscripts are within the most recently calculated subscript bounds, i.e. if the value of <subexplist1> <u>op1</u> <intbplist1> is <u>true</u> (see also T 5.34 and T 5.35), is the value of the subscripted variable (<id1><βγs1>[<subexplist1>]) equal to the value of the same variable in the previous activation (<id1><βγs2>[<subexplist1>]).

Now if we want to know the value of a subscripted variable, no matter whether it corresponds to a normal or to an own array, we again apply the same kind of rules as with simple variables:

1. <id1>[<subexplist1>] is supplied with the current bn; moreover, <subexplist1> is evaluated, see T 5.40, T 5.38, i.e. we continue with the evaluation of <id1><βγs1> [<u>va</u> {<subexplist1>}].
2. if we arrive at truth T 5.44 obviously no assignment was performed to this subscripted variable in the block in which its corresponding array identifier was declared, so its value is ω.
3. the next possibility is that <id1><βγs1> is a formal array identifier, which by applying T 5.43 is then replaced by its actual identifier and supplied with the bn of the block in which the procedure statement occurs.

4. if neither 2 nor 3 are applicable we try the smallest embracing block.
5. when there is no longer a smallest embracing block, the value of the subscripted variable is ω.

7. Switch declarations (T 6.1 to T 6.13)

The prescan rule for determining the value of
<βγs1><αs1>: switch <id1>:= <dexplist1>;≤decl list1≥ ≤st list1≥ end
is again similar to truth T 4.9 for type declarations.

1. In phase 1 of the prescan we add to V switch <id1><βγs1>, provided that <id1> has not been declared before in the same block.
2. In phase 2 of the prescan we see whether the switch list contains only declared identifiers.
3. In phase 1 of the execution we define a value for τ<βγs1><αs1> and continue with the evaluation of τ<βγs1><αs1>α .
4. Only in phase 2 of the execution we really have to do something, which is described in truths T 6.9 to T 6.13.

We give an example:
After applying these rules to a switch declaration of the form
switch S := L, if i > 0 then P else Q, M[3]
and assuming that the current bn is <βγs1>≤δβγs1≥ the result is that to V is added:

a. switch S <βγs1>≤δβγs1≥ co
b. S < βγs1> [<subexp>] is undefined switch designator co
c. S <βγs1>[1] is L co
d. S <βγs1>[2] is if i > 0 then P else Q co
e. S <βγs1>[3] is M[3] co.

The way in which we use these results will become clear when the definition of goto statements is given.

8. Procedure declarations (T 7.1 to T 7.50)

The prescan rules for a procedure declaration are given in truths T 7.29, T 7.30. We distinguish between procedure declarations with formal parameters and without.

1. A procedure declaration without formal parameters.
   - a. In phase 1 of the prescan we note the procedure identifier and check whether it has not been declared before in the same block.
   - b. In phase 2 of the prescan we do the following:
     - b1. we evaluate: begin integer dummy; <st1> end in <decl block>. <st1> is the procedure body which is made into a block by surrounding it with begin integer dummy; ....end.

     Apparently we here again initialize the prescan of this newly created block (see T 1.3).
     - b2. The first label of the procedure body is stored.
     - b3. A new rule is given for the evaluation of <βγs1><αs1> .
     - b4. The value of <βγs1><αs1>α is determined.
   - c. In phase 1 of the execution we define the value of τ<βγs1><αs1> and continue by evaluating <βγs1><αs1>α.
   - d. In phase 2 of the execution the procedure identifier is supplied with the current bn and with the first label of its procedure body which can be found because of b2.

     Thereafter we continue by evaluating τ<βγs1><αs1>α.

2. A procedure declaration with formal parameters.
   - a. In phase 1 of the prescan we note the procedure heading (this makes it possible to check in phase 2 of the prescan whether a procedure statement has the correct number of actual parameters), and check whether the procedure identifier has not been declared before in the same block.
   - b. In phase 2 of the prescan we do the following:
     - b1. we evaluate an extra begin.
     - b2. we evaluate formal <idlist1> , where <idlist1> is the list of formal parameters.

       The effect of evaluating for example formal f,g assuming that the current bn is <βγs1> , is simply that we add to V:

formal f <βγs1> and also formal g < βγs1>.
These notes are used later on to check whether a statement contains only declared identifiers.

b3. we evaluate begin integer dummy; <st1> end in <decl block>, where <st1> is the procedure body.

b4. we store the first label of the procedure body.

b5. we evaluate the end, corresponding to the extra begin in b1.

c. as in 1c.

d. In phase 2 of the execution those entries in the formal parameter list which occur in the value part are supplied with a special indication, which is simply the corresponding specifier.

Furthermore, two rules are added to V:

d1. the procedure identifier supplied with the current bn and the first label of the procedure body.

d2. the procedure identifier supplied with the current bn and the formal parameter list extended with the indications mentioned above.

Finally we continue by evaluating τ<βγs1><αs1>α.

Remarks:

1. Only <type> , <type> array and <type> procedure are allowed as specifiers of value formal parameters.
2. Specifications of non value parameters are ignored.

9. Assignment statements (T 8.1 to T 8.46)

The ultimate result of the evaluation of an assignment statement is that we add to V:
the name of the variable concerned, followed by the bn of the block in which it has been declared, followed by is, followed by the value of the expression in the right hand side, e.g.

A βγ ββγ is 10 or B βγ βγ ββγ is true.

The details are demonstrated by the following example:

Suppose we want to evaluate:

A := f := M[i] := g[p,q] := <exp1> (1)

where f and g are formal parameters and suppose that the current bn is <βγs1>. Then (1) is changed into

A <βγs1> := f := M[i] := g[p,q] := <exp1> . (2)

Suppose A is a variable of type integer which was declared in a block with bn <βγs2> (where <βγs1> = <βγs2><βγs> , i.e., A has been declared in an embracing block).

Then (2) is changed into

f := M[i] := g[p,q] := integer A <βγs2> := <exp1> . (3)

Let us suppose that f has as its corresponding actual parameter the identifier b and that the procedure statement in which this assignment statement occurs, itself occurs in a block with bn <βγs3>.

Then (3) is changed into

b <βγs3> := M[i] := g[p,q] := integer A <βγs2> := <exp1> . (4)

(The reason we supply b with <βγs3> is that we want to avoid clash of names.)

Suppose b was declared integer in a block with bn <βγs4> , where <βγs3> = <βγs4><βγs>.

Then (4) is changed into

M[i] := g[p,q] := integer A <βγs2> := integer b <βγs4> := <exp1>.(5)

Next (5) is changed into

M <βγs1>[va{i}]:=g[p,q] := integer A <βγs2> := integer b <βγs4>:= <exp1> . (6)

If the value of i is 10 and if we find in V an entry of the form

integer array M <βγs 5> [1:12]

then (assuming that <βγs1> = <βγs5><βγs>), we check whether

1 ≤10 ∧ 10 ≤12 is true (i.e., 10 op1 1:12 is evaluated, see also T 5.34, T 5.35), and (6) is changed into

g[p,q] := integer A <βγs2> := integer b <βγs4> := integer array M <βγs5> [10] := <exp1> . (7)

Then (7) is changed into

If the actual array identifier corresponding to g is B and if p and q have the values 0 and 1 then (8) is changed into

B <βγs3>[0,1]:= integer A <βγs2>:= integer b<βγs4> := integer array M <βγs5>[10] : = <exp1> (9)

(9) is treated similarly to (6) and so next we have to evaluate

integer A <βγs2> := integer b <βγs4> := integer array M <βγs5>[10] := integer array B <βγs6>[0,1]:= <exp1> . (10)

Now <exp1> is evaluated (see T 8.30), and (10) is changed into, say

integer A <βγs2> := integer b <βγs4> := integer array M <βγs5>[10] := integer array B <βγs6>[0,1] := 87.

This is changed into the evaluation of successively:

integer A <βγs2> := 87, integer b <βγs4> := 87,
integer array M <βγs5>[10] := 87, integer array B <βγs6>[0,1]:= 87.

which finally leads to the following list of entries in V:

A <βγs2> is 87 co b <βγs4> is 87 co M <βγs5>[10] is 87 co B <βγs6>[0,1] is 87.

Remark: It was not necessary to treat assignment to procedure identifiers separately because in the evaluation of a type procedure, e.g. <id1> or <id1> (<actpalist>), of type <type1>, before entering the procedure body first an extra type declaration of the form <type1><id1> is introduced, see T 4.20, T 10.15, T 10.44 and T 10.45.

10. For statements (T 11.1 to T 11.15)

A for statement is essentially replaced by a list of equivalent simpler statements. However, two difficulties arise:

1. The requirement that the value of the controlled variable be undefined upon exhaustion of the forlist.
2. The requirement that the value of a goto statement leading into a for statement be undefined.

Concerning 1, it is of course easy to include in our system a rule defining the value of the controlled variable upon exit to be ω. However, because a for statement with more than one for list element is first rewritten as a list of for statements with just one for list element each, we had to make a distinction between the last statement of such a list and the earlier ones. This we did by introducing the symbol for 1 when the corresponding for list element was not the last one of the for list, for example:

for i:=1,2 do <st1>

is replaced by

for 1 i:=1 do <st1> ; for i:=2 do <st1>.

Problem 2 we solved by considering the for statement as a block during the execution phase, but not in the prescan since a label in a for statement may appear in a switch list.

This is described in more detail by an explanation of truths T 11.9 and T 11.11. The value of

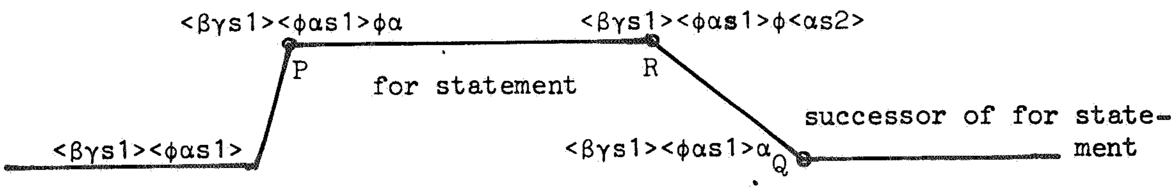
<βγs1> <ϕαs1> : for <intvar1> := <forlistel1> do <st1><specialstlist1> <end1>

is given by:

1. To V is added: <βγs1><ϕαs1> is {⫲<βγs1><ϕαs1> is {begin co ⫲τ<βγs1><ϕαs1> is τ<βγs1><ϕαs1>ϕα⫲ co <βγs1><ϕαs1>ϕα}⫲co <βγs1><ϕαs1>ϕα} .

So if in phase two of the prescan we ask for the value of <βγs1><ϕαs1> we add a new rule for evaluating <βγs1><ϕαs1> to V and continue by evaluating <βγs1><ϕαs1>ϕα.

Here the evaluation of <βγs1><ϕαs1>ϕα leads into the for statement as may be illustrated by a picture:



Thus, the extra ϕ counts for statements.

In phase 1 of the execution we evaluate begin, define the value of τ <βγs1><ϕαs1> and continue with the evaluation of <βγs1><ϕαs1>ϕα.

In phase 2 of the execution we evaluate τ<βγs1><ϕαs1> which leads to evaluation of τ<βγs1><ϕαs1>ϕα.

2. We determine the value of

<βγs1><ϕαs1>ϕα : for begin <intvar1> := <forlistel1> ; <st1> forend

(see point P in the picture).

This is defined in truths T 11.13, T 11.14 and T 11.15.

3. We evaluate

<βγs1><ϕαs1>α: <specialstlist1> <end1>

(see point Q in the picture).

Furthermore, the value of

<βγs1><ϕαs1>ϕ<αs2> : for end

is given by writing the following in V:

<βγs1><ϕαs1>ϕ<αs2> is {ǂ<βγs1><ϕαs1>ϕ<αs2> is
{ǂτ<βγs1><ϕαs1>ϕ<αs2> is {end co τ<βγs1><ϕαs1>α}ǂ co <βγs1><ϕαs1>α}ǂ
co <βγs1><ϕαs1>α} .

Hence, if in phase 2 of the prescan we ask for the value of <βγs1><ϕαs1>ϕ <αs2> we add a new definition of <βγs1><ϕαs1>ϕ<αs2> to V and continue with the evaluation of <βγs1><ϕαs1>α (here we see the link between R and Q in the picture).

If in phase 1 of the execution we ask for the value of <βγs1><ϕαs1>ϕ <αs2> we first give a rule for evaluating τ<βγs1><ϕαs1>ϕ < αs2> and then continue with <βγs1><ϕαs1>α.

In phase 2 of the execution we evaluate end and continue with τ<βγs1><ϕαs1>α.

For the evaluation of

<βγs1><ϕαs1> : for begin <intvar1> :=<forlistel1> ; <st1> for end

we have to distinguish between the three possibilities for a for list element.

If the for list element is an arithmetic expression we determine the value of

<βγs1><ϕαs1> : <intvar1> :=<aexp1> ; <st1> for end.

This means that the for statement is simply rewritten as an assignment statement, followed by the statement which originally followed do.

Now it becomes clear why we had to introduce the metalinguistic variable <end> (end in <end> , forend in <end> , forend <intvar1> in

<end> ;
this way of adding the controlled variable to the forend is an indication that we have to give <intvar1> the value ω at the end of the processing of the for statement): we wanted a similar treatment of the evaluation of e.g.:

<βγs1><φαs1> : <st list1> end

and of

<βγs1><φαs1> : <st list1> forend.

If the for list element is a while element or a step until element we use the definitions in [1] , 4.6.4.2 and 4.6.4.3.

11. Procedure statements and function designators (T 10.1 to T 10.45)

We distinguish four cases.

1. A single identifier, occurring as a procedure statement where in the corresponding declaration the identifier was not defined as a type procedure (for the case that the identifier was declared as a type procedure we refer to T 10.44), i.e. something of the form st : <id1> , see also T 3.8. We search the embracing blocks until we find the corresponding declaration in the block with bn <βγs1> (if the identifier turns out to be formal we substitute the actual parameter and continue with the evaluation of st : <id2><βγs2>).
Then we evaluate:

a. enter procedure <βγs1> (see T 10.16).
b. the first label of the procedure body which was stored at the time of declaration (see T 7.29, T 7.37 and T 7.38).
c. exit procedure (see T 10.17).

2. A procedure identifier followed by a non empty actual parameter part, occurring as a procedure statement; i.e. something of the form st : <id1> (<actpalist1>). We again search the embracing blocks for the corresponding declaration (we perform the substitution of an actual parameter for the formal <id1> if necessary) which will look like <id1><βγs1> (extformallist1).
Then we evaluate

a. **enter procedure** <βγs1>

b. <id1><βγs1> (<extformallist1>) **op8** <id1><βγs1> (<actpalist1>), this leads to the formal actual substitution and the evaluation of the first label of the procedure body (see below for more details),

c. **exit procedure**.

Remark: For the case that the procedure was declared as a type procedure we refer to 10.45.

3. A single identifier occurring in an expression, where the corresponding declaration defines the value of a function designator and has the form <type1> **procedure** <id1><βγs1> **is** <βγs2>α.
Then we evaluate

a. **enter procedure** <βγs1>	
b. **begin**	a fictitious block is introduced,
c. <type1> <id1>	a type declaration for the procedure identifier is given in this new block,
d. <βγs2>α	the procedure body is executed,
e. **function value** **op15** <id1>	the value assigned to the procedure identifier is stored
f. **end**	end of the fictitious block,
g. **exit procedure**	
h. **function value**	finally the value of the function designator is equal to the value of **function value**.

This mechanism is capable of handling recursive function designators.

4. A procedure identifier followed by a non empty actual parameter part occurring in an expression. This is treated similarly to case 3 except for a change in d. For now we first have to do the formal actual substitution and after this we perform the evaluation of the first label of the procedure body.

Next we give some details concerning the formal actual substitution.

1. A formal parameter which is called by value and was specified **integer** or **boolean**.

We evaluate

a. begin — we enter a fictitious block,

b. <type1><id1> — declaration in the fictitious block,

c. <type1><id1> op9 <actpa1> — this resembles an assignment statement with some precautions because of the possibility of clash of names,

d. the formal actual substitution of the remaining parameters, if necessary,

e. the first label of the procedure body,

f. end — we leave the fictitious block.

2. A formal parameter <id1> which is called by name :

a. If there are formal parameters left in the extended formal list which are called by value they are treated first (see T 10.20).

b. Otherwise we evaluate

b1. begin — we enter a fictitious block,

b2. <id1> op10 <actpa1> — the effect is that a note is left in V to the effect that <actpa1> is the corresponding actual of <id1>; moreover, <id1> is supplied with the current bn and <actpa1> with the bn of the block in which the procedure is called,

b3. The formal actual substitution of the remaining parameters, if necessary.

b4. The first label of the corresponding procedure body,

b5. end — we leave the fictitious block.

3. A formal parameter which was called by value and was specified integer procedure or boolean procedure is treated like a formal parameter called by value and specified integer or boolean.

4. Value arrays.

a. a new block is created.

b. the declaration of the actual array identifier is looked up.

c. the formal parameter is declared to be an array with the same type and the same bound pair list as the actual parameter.

d. the value of the actual array identifier (i.e. the value of the corresponding array of subscripted variables) is assigned to the newly declared array (truths T 10.40 to T 10.43).

e. the formal actual substitution of the remaining parameters is performed, if necessary.

f. we leave the block which was created in a.

12. Goto statements (T 9.1 to T 9.26)

In truths T 9.1 to T 9.4 the case is treated in which the designational expression which occurs in the goto statement is not a label or a switch designator.

In truths T 9.5 to T 9.7 the value of a conditional goto statement is defined (after application of T 3.3 to T 3.5 every conditional statemement has been rewritten as a sequence of other statements in which the only conditional statements are if statements of the form

if <bexp> then goto <dexp>).

In the evaluation of if <bexp1> then goto <dexp1> (1)

we distinguish three cases:

1. <bexp1> is equal to the symbol false.

Then we add true to V. It is clear that this has no influence on the evaluation of the rest of the program so here we have the equivalent of the dummy statement (except for the fact that possible side effects of the evaluation of the boolean expression which turned out to have the value false are not cancelled).

2. <bexp1> is equal to the symbol true. Then the value of (1) is equal to the value of goto <dexp1>.

3. Neither 1 nor 2 holds. Then we evaluate

if va {<bexp1>} then goto <dexp1> .

Two possibilities arise:

3.1. <bexp1> has the value false or true. Then in the next step 1 or 2 will apply.

3.2. <bexp1> has some other value (for example, if <bexp1> was a formal parameter which was called by name with an integer as corresponding actual parameter). Then only truth T 1.1 applies and the value of the whole if statement is ω.

The evaluation of goto <label1> is governed by the following five rules:

1. <label1> is supplied with the current bn and we continue the evaluation with goto <label1><βγs1>.
2. By repeated application of the rule
   goto <label1><βγs1><βγ> is goto <label1><βγs1>
   (including the case that we do not need to apply it), we search the embracing blocks until one of the following three possibilities occurs:
3. In V we find a rule of the form label <label1><βγs1>≤δβγs1≥ is τ<βγs2><ϕαs1> (such a rule was entered in V as a result of the application of T 3.15).
   Then, first <βγs1>≤δβγs1≥ is added to V; hence, the bn of the block in which <label1> was declared (with an obvious extension of the meaning of declared) is activated. Next we evaluate τ<βγs2><ϕαs1> which leads to the continuation of the execution of the program with the statement which was labelled by <label1>.
4. In V we find a rule of the form
   <id1><βγs1> op2 (<dexp1>/<βγs2>≤δβγs1≥).
   Then, in order to evaluate goto <id1><βγs1> (apparently <id1> is a formal parameter) we do the following:
   4.1. save bn <id1><βγs1>; this was defined in T 4.22.
   4.2. the bn of the block in which the procedure statement containing goto <id1> occurred is activated,
   4.3. goto <dexp1> is evaluated.
   4.4. the bn which was saved in 4.1 is restored.
   In order to avoid clash of names we need 4.2 but because goto <dexp1> might prove to be equivalent to the dummy statement we have to take the precautions 4.1 and 4.4.
5. Neither 3 nor 4 applies. Then finally we have to evaluate goto <label1>βγ : there is no longer an embracing block and the value of the goto statement is ω.

Finally we consider the case where the designational expression is a switch designator, say <id1>[<subexp1>].
We supply the switch identifier with the current bn, evaluate the subscript expression and search the embracing blocks until we find the one in which the switch identifier had been declared; i.e., supposing we were evaluating goto <id1><βγs1>[<subexp1>] until we find in V: switch <id1><βγs1><δβγs1> (the case of a formal switch identifier is similar to a formal array identifier).
Then we do the following:

1. We save the current bn.
2. We activate <βγs1><δβγs1>.
This is necessary because of [1], 5.3.5.
3. We evaluate op13 <id1><βγs1> [<subexp1>] .
The result of the switch declaration was a list of the form:

<id1><βγs1> [<aexp>] is undefined switch designator co
<id1><βγs1>[1] is <dexp> co (1)
<id1><βγs1>[2] is <dexp> ,

etc. Now, if <subexp1> is less than 1 or greater than the number of items in the switch list the effect of op13 <id1><βγs1> [<subexp1>] is that we add to V:

<id1><βγs1>[<subexp1>] is {undefined switch designator}.

Otherwise we distinguish between two cases:

3.1. The designational expression which stands at the right hand side of the symbol is in the list is a label or a switch designator (here of course we select that item in the list (1) where the value of the subscript in the left hand side is equal to <subexp1> in op13 <id1><βγs1> [<subexp1>]).
Then we add to V (after an intermediate introduction of op14)
<id1><βγs1> [<subexp1>] is {<dexp1>} .

3.2. <dexp1> is not a label or a switch designator.
This case is reduced to 3.1 by application of T 9.20 to T 9.23.

4. We reactivate the bn which was saved in 1.
5. We evaluate
goto va {<id1><βγs1>[<subexp1>]},
where the value of <id1><βγs1>[<subexp1>] was written in V as a

result of application of the rules explained in 3. For example goto va {<id1><βγs1>[0]} leads to the evaluation of goto undefined switch designator.
This has the effect that true is added to V.
Again this does not influence the evaluation of the rest of the program (with the same reservations as with the conditional goto statement).

13. Expressions (T 12.1 to T 17.69)

In truths T 12.1 to T 12.90 we give the syntactic definitions of

<exp>	and <decl exp> ,
<int varid>	and <decl int varid>,
<boolean varid>	and <decl boolean varid>,
<int array id>	and <decl int array id>,
<boolean array id>	and <decl boolean array id>,
<subexplist>	and <decl subexplist>,
<int var>	and <decl int var>,
<boolean var>	and <decl boolean var>,
<actpa>	and <decl actpa>,
<int functdes>	and <decl int functdes>,
<boolean functdes>	and <decl boolean functdes>.

We use the information which was left in V as a result of the prescan and apply the standard technique for the search in embracing blocks.

In truths T 13.1 to T 13.30 we define syntactically <aexp> and <decl aexp>. Here we use the definitions of <in>, <int var>, <decl int var>, <int functdes>, <decl int functdes>, <bexp> and <decl bexp>.
In truths T 14.1 to T 14.40 we define syntactically <bexp> and <decl bexp>.
Here we use the definitions of
<logical value>, <boolean var>, <decl boolean var>, <boolean functdes>, <decl boolean functdes>, <saexp> and <decl saexp>.
In truths T 15.1 to T 15.26 we define the syntax of designational

expressions.
In truths T 16.1 to T 16.7 we define syntactically the value of:
a conditional arithmetic expression,
a conditional boolean expression,
an arithmetic expression between parentheses,
a boolean expression between parentheses.

In truths T 16.8 to T 16.40 we define the value of a boolean expression. First we give some general truths concerning expressions involving variables and function designators or expressions containing more than one operator, then we define the truth-tables for the logical operators and finally we treat relations involving integers:
every relation is reduced to the relation <in>≤<in> which in turn is reduced to the evaluation of <in> ≤0 which is defined in T 16.38 to T 16.40.

Truths T 16.41 and T 16.42 are useful in the evaluation of e.g. +(-3); truths T 16.43 to T 16.45 we use in the evaluation of e.g. 3+(-5) and +3 ≤ +5.
Truths T 16.46 to T 16.49 define the value of an arithmetic expression involving integer variables or function designators or expressions containing more than one operator.
The special form of T 16.46 was introduced to treat a case like (-2)↑(+2): using the obvious form of truth T 16.46 this would have been equivalent to -2 ↑ 2 which is wrong.
Truths T 16.50 to T 16.57 define exponentiation, integer division and multiplication.
Truths T 16.58 to T 16.98 define addition and subtraction.

The value of a part of the program which was left undefined, said to be undefined or forbidden was ω. The effect of truth T 17.68 is that in such a case the value of the whole program is ω.

14. A conclusion

We note that the description in Chapter 3 has been lengthened and complicated considerably by:

1. The requirement that all identifiers of a block be declared, even in sections of the program which are not executed.
2. The requirement that no identifier be declared more than once in any one block head.
3. The requirement that the effect of a goto statement, leading into a for statement, be undefined.
4. The requirement that the value of the controlled variable be undefined upon exhaustion of the for list.
5. The possibility of clash of names, cf. the last sentences of [1], 4.7.3.2 and 4.7.3.3 and also [1] , 5.3.5.
6. The requirement that the effect of a goto statement leading to an undefined switch designator be equivalent to the dummy statement.
7. The requirement that it be possible that a type procedure is used as a statement.

It seems desirable that one not burden ALGOL X unnecessarily by these and similar requirements.

15. List of truths which need some special explanation

T 1.2 <program1> which may be a compound statement or may be labelled is transformed into
<u>begin</u> <u>integer</u> <u>dummy</u>; <program1> <u>end</u>,
the bn is set to $\beta\gamma$, the prescan is performed and by asking for the value of $\beta\gamma\alpha$ the program is executed.

T 1.24 - T 1.25 see also for statement.

T 2.1 - T 2.33 some special measures were taken for the inclusion of the dummy statement because the mechanism of the processor cannot handle a truth of the form "<u>in</u> <unlabelledbasic st> ". Hence, we have to use the optional brackets <u><and></u> in places where a dummy statement might occur, e.g. T 2.20, T 2.23, T 2.30 etc.

T 3.8 the procedure statement is supplied with <u>st</u>: in view of the possibility that <procst1> is a type procedure, in which case we know later on that we are not interested in the function value.

T 3.11 <βγs1><βγ1>was left in V as a result of the evaluation of <block1> in <decl block>. However, at the moment of application of T 3.11 it is not the current bn since this has been reset to <βγs1>.

T 4.1 a case like .... integer i,j; ... is transformed into ... integer i; integer j; ... .

T 4.20 see also procedure statements.

T 5.14 cf. T 4.1.

T 7.26 we check (during the prescan) whether the number of actual parameters is equal to the number determined by the formal parameter list of the corresponding declaration.

T 7.37 cf. T 3.11.

T 7.41 - T 7.47 an extended formal list is a list of formal parameters each of which may be extended with an indication that it occurs in the value part.

T 7.48 we ignore the specifications.

T 8.31 this truth applies if T 8.43 to T 8.46 do not apply, e.g. in the evaluation of boolean <id><βγs>:= <in> . (1)
We use it in order to avoid infinite application of T 8.27 in a case like (1).

T 10.5, T 10.10 see also T 10.44 and T 10.45.

T 10.20 we first perform the call by value of the remaining parameters: <ext formal list1> is not simply a <idlist> , otherwise we would have had to apply T 10.21 first.

T 10.25 cf. T 8.31.

T 11.9, T 11.10 the only difference between these truths is the addition of <intvar1> to forend in T 11.9.

T 11.11, T 11.12 the only difference between these truths is that we set <intvar1> to ω in T 11.12.

T 12.89, T 12.90 see also T 7.26.

T 16.1 - T 16.7 the corresponding definitions for designational expressions are given in T 9.1 to T 9.4 and T 9.20 to T 9.23.

T 17.56 cf. T 11.15.

T 17.57 cf. T 1.2, T 7.29 and T 7.30.

T 17.58, T 17.59 cf. T 3.3 to T 3.5, T 11.14 and T 11.15.

Chapter 3

In this chapter we give the formal definition of ALGOL 60. For typographical reasons we introduced some changes in the notation:

(denotes {

) denotes }

: denotes ÷

⊐ denotes ⊃

a denotes α

b denotes β

c denotes γ

d denotes δ

f denotes φ

l denotes λ

m denotes μ

o denotes ω

t denotes τ

Thus, e.g., bc should be read as βγ and not as an independent basic symbol, created by underlining.

The numbers to the left of the truths are not to be interpreted as a part of the truths; they are used only for reference purposes.

Errata

T 1.18 should read:

begin <βγs1><δβγs1> is ∤ <βγs1>βγ <δβγs1> ∤ co

T 1.21 should read:

end βγ is ∤ end βγ is ∤ <name> ∤∤ co

T 9.17 should read:

switch <id1><βγs1><δβγs1> → goto <id1><βγs1>[<subexp1>]

is

{save bn <id1><βγs1> co ⦃<βγs1><δβγs1> ⦄ co
op13 <id1><βγs1>[<subexp1>] co goto va {<id1><βγs1>[<subexp1>]}
co reset bn <id1><βγs1>} co

T 15.14 should read:

switch <id1><βγs1> → <id1><βγs1> in <decl switchid> co .

On page 48 items 4 and 5 should be interchanged.

1.1 <name> is ⊰ o ⊱ co

1.2 <program1> is
(⊰ bc ⊱ co bca : integer dummy; <program1> end co bca) co

1.3 begin <decllist1> <stlist1> end in <declblock> is
(begin co <decllist1> <stlist1> end) co

1.4 <bcs1> --> <decllist1> < stlist1> end is
(⊰ <bcs1>a is (⊰ <bcs1>a is (begin co
⊰ t <bcs1>a is t <bcs1>aa ⊱ co <bcs1>aa) ⊱ co <bcs1>aa) ⊱ co
<bcs1>aa : <decllist1> <stlist1> end) co

1.5 <bcs1><fas1> : end is
(⊰ <bc1><fas1> is (end co ⊰ <bcs1><fas1> is
(⊰ t <bcs1><fas1> is end ⊱ co t<bcs1>a) ⊱ co) ⊱ co <bcs1>a) co

1.6 a in <as> co
1.7 a<as> in <as> co

1.8 <as> in <fas> co
1.9 <as>f<fas> in <fas> co

1.10 b in <bs> co
1.11 b<bs> in <bs> co

1.12 <bs>c in <bc> co

1.13 <bc> in <bcs> co
1.14 <bc><bcs> in <bcs> co

1.15 d <bcs> in <dbcs> co
1.16 d<bcs><dbcs> in <dbcs> co

1.17 <bcs1><dbcs1> --> begin is begin <bcs1><dbcs1> co

1.18 begin <bcs1><dbcs1> is ⊰ <bcs1>bc <dbcs1> ⊱ co

1.19 <bcs1><bc1><dbcs> -->
begin <bcs1><dbcs1> is ⊰ <bcs1>b <bc1><dbcs1> ⊱ co

1.20 <bcs1><dbcs1> --> end is end <bcs1><dbcs1> co

1.21 end bc is < end bc is ⊰ <name> ⊱ ⊱ co

1.22 end <bcs1><bc1><dbcs1> is ⊰ <bcs1><dbcs1> ⊱ co

1.23 end in <end> co

1.24 forend in <end> co
1.25 forend <intvar> in <end> co

```
2.1   <typedeclaration>                 in  <declaration> co
2.2   <arraydeclaration>                in  <declaration> co
2.3   <switchdeclaration>               in  <declaration> co
2.4   <proceduredeclaration>            in  <declaration> co

2.5   <declaration> ;                   in  <decllist> co
2.6   <decllist><declaration> ;         in  <decllist> co

2.7   <assst>          in  <unlabelledbasicst> co
2.8   goto <dexp>      in  <unlabelledbasicst> co
2.9   <procst>         in  <unlabelledbasicst> co

2.10  <unlabelledbasicst>               in  <basicst> co
2.11  <label> :                         in  <basicst> co
2.12  <label> : <basicst>               in  <basicst> co

2.13  <block>          in  <uncst> co
2.14  <compoundst>     in  <uncst> co
2.15  <basicst>        in  <uncst> co

2.16  <uncst>   in  <st> co
2.17  <condst>  in  <st> co
2.18  <forst>   in  <st> co

2.19  <st>             in  <stlist> co
2.20  <st> ; <stlist>  in  <stlist> co

2.21  ; < stlist>      in  <specialstlist> co

2.22  begin  <stlist>  end          in  <unlabelledcompound> co

2.23  <unlabelledcompound>          in  <compoundst> co
2.24  <label> : <compoundst>        in  <compoundst> co

2.25  begin  <decllist> < stlist>  end    in  <unlabelledblock> co

2.26  <unlabelledblock>   in  <block> co
2.27  <label> : <block>   in  <block> co

2.28  <block>          in  <program> co
2.29  <compoundst>     in  <program> co

2.30  if  <bexp>  then  <uncst>         in  <condst> co
2.31  if  <bexp>  then  <forst>         in  <condst> co

2.32  if  <bexp>  then  <uncst>  else  <st>    in  <condst> co
2.33  <label>  : <condst>                      in  <condst> co
```

3.1 <bcs1><fas1> : ; < stlist1><end1> is
<bcs1><fas1> : < stlist1><end1> co

3.2 <bcs1><fas1> : begin < stlist1> end <specialstlist1> <end1> is
<bcs1><fas1> : < stlist1> < specialstlist1> <end1> co

3.3 <bcs1><fas1> : if <bexp1> then < uncst1> <specialstlist1> <end1> is
<bcs1><fas1> : if ¬ (<bexp1>) then goto <bcs1><fas1>l ;
<uncst1>; <bcs1><fas1>l : <specialstlist1><end1> co

3.4 <bcs1><fas1> : if <bexp1> then < forst1> <specialstlist1><end1> is
<bcs1><fas1> : if ¬ (<bexp1>) then goto <bcs1><fas1> l ;
< forst1> ; <bcs1><fas1>l: <specialstlist1><end1> co

3.5 <bcs1><fas1> : if <bexp1> then < uncst1> else <st1>
< specialstlist1> <end1> is
<bcs1><fas1> : if ¬ (<bexp1>) then goto <bcs1><fas1>l ;
<uncst1> ; goto <bcs1><fas1>m ; <bcs1><fas1>l :
<st1> ; <bcs1><fas1>m : <specialstlist1><end1> co

3.6 <bcs1><fas1> : if <bexp1> then goto <dexp1><specialstlist1><end1> is
(≮ <bcs1><fas1> is (<bexp1> in <decl bexp> co <dexp1> in <decl dexp> co
≮ <bcs1><fas1> is (≮ t <bcs1><fas1> is
(if <bexp1> then goto <dexp1> co t <bcs1><fas1>a) ≯ co
<bcs1><fas1>a) ≯ co <bcs1><fas1>a) ≯ co
<bcs1><fas1> a : < specialstlist1> <end1>) co

3.7 <bcs1><fas1> : goto <dexp1> < specialstlist1> <end1> is
(≮ <bcs1><fas1> is (<dexp1> in <decl dexp> co
≮ <bcs1><fas1> is (≮ t<bcs1><fas1> is
(goto <dexp1> co t <bcs1><fas1> a) ≯ co <bcs1><fas1> a) ≯ co
<bcs1><fas1> a) ≯ co <bcs1><fas1>a : <specialstlist1> <end1>) co

3.8 <bcs1><fas1> : <procst1> < specialstlist1> <end1> is
(≮ <bcs1><fas1> is (<procst1> in <decl procst> co
≮ <bcs1><fas1> is (≮ t <bcs1><fas1> is
(st : <procst1>co t <bcs1><fas1>a) ≯ co <bcs1><fas1> a) ≯ co
<bcs1><fas1> a) ≯ co <bcs1><fas1>a : <specialstlist1> <end1>) co

3.9 <bcs1><fas1> : <assst1> < specialstlist1> <end1> is
(≮ <bcs1><fas1> is (<assst1> in <decl assst> co
≮ <bcs1><fas1> is (≮ t <bcs1><fas1> is
(<assst1> co t <bcs1><fas1>a) ≯ co <bcs1><fas1>a) ≯ co
<bcs1><fas1>a) ≯ co <bcs1><fas1>a : <specialstlist1><end1>) co

3.10 <bcs1><fas1> : <block1> <specialstlist1> <end1>
is
({ <bcs1><fas1> is (<block1> in <declblock> co
firstlabelofblock <bcs1><fas1> co { <bcs1><fas1> is
({ t<bcs1><fas1> is (firstlabelofblock <bcs1><fas1> co
t <bcs1><fas1>a) } co <bcs1><fas1>a) } co <bcs1><fas1>a) } co
<bcs1><fas1> a : <specialstlist1> <end1>) co

3.11 <bcs1><bc1> --> firstlabelofblock <bcs1><fas1> is
{ firstlabelofblock <bcs1><fas1> is <bcs1><bc1>a } co

3.12 <bcs1><fas1> : <label1> : < stlist1> <end1>
(label <label1><bcs1> co { <bcs1><fas1> is ({ <bcs1><fas1> is
(<label1> op3 <bcs1><fas1>a co { t <bcs1><fas1> is t<bcs1><fas1>a } co
<bcs1><fas1>a) } co <bcs1><fas1>a) } co
<bcs1><fas1>a : < stlist1> <end1>) co

3.12 label <label1><bcs1> is { label <label1><bcs1> } co

3.13 <specifier> <label1><bcs1> --> label <label1><bcs1> is o co

3.14 <bcs1><dbcs1> --> <label1> op3 <bcs2><fas1> is
{ label <label1><bcs1><dbcs1> is t <bcs2><fas1> } co

4.1 <bcs1><fas1> : <localorowntype1><id1>,<idlist1>;
<decllist1> <stlist1> end is
<bcs1><fas1>:<localorowntype1><id1>;<localorowntype1><idlist1>;
<decllist1> < stlist1> end co

4.2 integer in <type> co
4.3 boolean in <type> co

4.4 <type> in <localorowntype> co
4.5 own <type> in <localorowntype> co

4.6 <id> in <idlist> co
4.7 <id>,<idlist> in <idlist> co

4.8 <localorowntype><idlist> in <typedeclaration> co

4.9 <bcs1><fas1> : <type1> <id1> ; <decllist1> <stlist1> end is
(<type1><id1><bcs1> co ⊀ <bcs1><fas1> is (⊀ <bcs1><fas1> is
(⊀ t <bcs1><fas1> is (<type1><id1> co t <bcs1><fas1>a) ⊁ co
<bcs1><fas1>a) ⊁ co <bcs1><fas1>a) ⊁ co
<bcs1><fas1> a : < decllist1> <stlist1> end) co

4.10 <bcs1><dbcs> --> <type1><id1> is ⊀ <type1><id1><bcs1> ⊁ co

4.11 <type1><id1><bcs1> is < <type1><id1><bcs1> ⊁ co

4.12 <specifier><id1><bcs1> --> <type><id1><bcs1> is o co

4.13 <bcs1><fas1> : own <type1><id1>; <decllist1> <stlist1> end is
(<type1><id1><bcs1>co ⊀ <bcs1><fas1> is (⊀ <bcs1><fas1> is
(⊀ t <bcs1><fas1> is (<bcs1><fas1>: own <type1><id1>co
t <bcs1><fas1>a) ⊁co<bcs1><fas1>a) ⊁ co <bcs1><fas1>a) ⊁ co
<bcs1><fas1>a : < decllist1> <stlist1> end) co

4.14 <bcs1><dbcs> --> <bcs2><fas1> : own <type1><id1> is
(⊀ <type1><id1><bcs1> ⊁ co ⊀ t <bcs2><fas1> is
(<bcs2><fas1> : own <type1><id1><bcs1>co t <bcs1><fas1>a) ⊁) co

4.15 <bcs1><dbcs> --> <bcs2><fas1> : own <type1><id1><bcs3> is
(⊀ <type1><id1><bcs1>⊁ co ⊀ <id1><bcs1> is <id1><bcs3> ⊁ co
⊀ t <bcs2><fas1> is
(<bcs2><fas1> :own <type1><id1><bcs1> co t <bcs1><fas1>a) ⊁) co

4.16 <bcs1><dbcs> --> <id1> is <id1><bcs1> co

4.17 <id>bc is o co

4.18 <id1><bcs1><bc> is <id1><bcs1> co

4.19 <id1><bcs1> op2 (<expl> / <bcs2><dbcs1>) -->
<id1><bcs1> is (savebn <id1><bcs1> co ≮ <bcs2><dbcs1> ≯ co
<id1><bcs1> op15 <expl> co resetbn <id1><bcs1> co
result <id1><bcs1>) co

4.20 (<type1> procedure <id1><bcs1> is <bcs2>a) -->
<id1><bcs1> is
(enterprocedure <bcs1> co begin co <type1><id1> co <bcs2>a co
functionvalue op15 <id1> co end co exitprocedure co functionvalue) co

4.21 <type><id1><bcs1> --> <id1><bcs1> is o co

4.22 <bcs1><dbcs1> --> savebn <id1><bcs2> is
≮ resetbn <id1><bcs2> is ≮ <bcs1><dbcs1> ≯ ≯ co

4.23 <id1><bcs1> op15 <expl> is <id1><bcs1> op15 va (<expl>) co

4.24 <id1><bcs1> op15 <in1> is ≮ result <id1><bcs1> is <in1> ≯ co

4.25 <id1><bcs1> op15 <logicalvalue1> is
≮ result <id1><bcs1> is<logicalvalue1> ≯ co

4.26 functionvalue op15 <id1> is functionvalue op15 va <id1> co

4.27 functionvalue op15 <in1> is ≮ functionvalue is <in1> ≯ co

4.28 functionvalue op15 <logicalvalue1> is
≮ functionvalue is <logicalvalue1> ≯ co

5.1 <decl aexp> : <decl aexp> in <decl bpair> co
5.2 <aexp> : <aexp> in <bpair> co

5.3 <decl bpair> in <decl bplist> co
5.4 <bpair> in <bplist> co

5.5 <decl bpair>,<decl bplist> in <decl bplist> co
5.6 <bpair>, <bplist> in <bplist> co

5.7 <in> : <in> in <intbplist> co
5.8 <in> : <in>, <intbplist> in <intbplist> co

5.9 <id> [<bplist>] in <arraysegment> co
5.10 <id>,<arraysegment> in <arraysegment> co

5.11 <arraysegment> in <arraylist>co
5.12 <arraysegment>,<arraylist> in <arraylist> co

5.13 <localorowntype>array <arraylist> in <arraydeclaration> co

5.14 <bcs1><fas1> : <localorowntype1> array <arraysegment1>,
<arraylist1>;<decllist1> <stlist1> end
is
<bcs1><fas1> :<localorowntype1> array <arraysegment1> ;
<localorowntype1> array <arraylist1> ;<decllist1> <stlist1> end co

5.15 <bcs1><fas1> : <type1> array <idlist1> [<bplist1>] ;
<decllist1> <stlist1> end
is
(<type1> array <idlist1> [<bplist1>] <bcs1> co
≮ <bcs1><fas1> is (≮ <bcs1><fas1> is (≮ t <bcs1><fas1> is
(<type1> array <idlist1>[<bplist1>] co t<bcs1><fas1>a) ≯ co
<bcs1><fas1>a) ≯ co <bcs1><fas1>a) ≯ co
<bcs1><fas1>a : <decllist1> <stlist1> end) co

5.16 <type1> array <id1>,<idlist1>[<bplist1>] <bcs1>
is
(<type1> array <id1> [<bplist1>] <bcs1> co
<type1> array <idlist1> [<bplist1>] <bcs1>) co

5.17 <type1> array <id1> [<bplist1>] <bcs1> <bc1>
is
(<type1> array <id1><bcs1><bc1> co ≮ <bcs1> > co
<bplist1> in <decl bplist> co ≮ <bcs1><bc1> ≯) co

5.18 <type1> array <id1><bcs1> is ≮ <type1> array <id1><bcs1> ≯ co
5.19 <specifier><id1><bcs1> --> <type> array <id1><bcs1> is o co

```
5.20   <type1> array <idlist1> [<bplist1>]  is
       <type1> array <idlist1> [ va (  <bplist1> )  ] co

5.21   <type1> array <id1>,<idlist1> [ <intbplist1> ]     is
       (    <type1> array <id1> [<intbplist1>] co
        <type1> array <idlist1> [<intbplist1>] )  co

5.22   <bcs1> < dbcs1>  --> <type1> array <id1> [<intbplist1>]  is
       ≮  <type1> array <id1> <bcs1> [ <intbplist1>]  ≯ co

5.23   <bcs1><bc1> < dbcs1>   -->   <aexp1> : <aexp2>  is
       (  ≮  <bcs1> <dbcs1> ≯ co  bpair  op15  <aexp1> : <aexp2> co
        ≮  <bcs1><bc1> <dbcs1>  ≯  co  bpair )  co

5.24   bpair  op15  <aexp1> : <aexp2> is
       bpair  op15  va ( <aexp1> )  :  va  (  <aexp2> )  co

5.25   bpair  op15  <in1> : <in2>  is ≮  bpair is<in1> : <in2> ≯ co

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

5.27   <intbplist1>  is  ≮  <intbplist1> ≯ co

5.28   <bcs1><fas1> : own <type1> array <idlist1> [<bplist1>] ;
                             <  decllist1> < stlist1>  end
       is
       (  <type1> array <idlist1>[<bplist1>] <bcs1> co
        ≮  <bcs1><fas1> is  (  ≮  <bcs1><fas1>  is (  ≮  t<bcs1><fas1> is
          (  <bcs1><fas1> : own <type1> array <idlist1> [<bplist1>] co
         t<bcs1><fas1> a ) ≯ co <bcs1><fas1>a )  ≯ co  <bcs1><fas1>a )  ≯ co
         <bcs1><fas1>a : < decllist1> < stlist1>  end  )  co

5.29   <bcs1><fas1> : own <type1> array <idlist1> [ <bplist1> ]  is
       <bcs1><fas1> : own <type1> array <idlist1> [  va ( <bplist1> )  ] co

5.30   <bcs3><dbcs> -->
       <bcs1><fas1> : own <type1> array <idlist1>[<intbplist1>] < bcs2>
       is
       ( ≮  t <bcs1><fas1> is
          <bcs1><fas1> : own <type1> array <idlist1>[<intbplist1>]<bcs3>≯ co
        own <type1> array <idlist1> [<intbplist1>] < bcs2>  )  co

5.31   own <type1> array <id1>,<idlist1> [ <intbplist1>] <bcs1>  is
       (  own <type1> array <id1> [ <intbplist1> ] <bcs1>  co
        own <type1> array <idlist1> [ <intbplist1> ]  <bcs1> )  co
```

5.32 <bcs1>< dbcs> --> own <type1> array <id1> [<intbplist1>] is
⊀ <type1> array <id1><bcs1> [<intbplist1>] ⊁ co

5.33 <bcs1><dbcs> --> own <type1> array <id1> [<intbplist1>]<bcs2>
is
(⊀ <type1> array <id1><bcs1> [<intbplist1>] ⊁ co
⊀ <subexplist1> op1 <intbplist1> -->
<id1><bcs1>[<subexplist1>] is <id1><bcs2>[<subexplist1>] ⊁) co

5.34 <in1> ≤ <in3> ∧ <in3> ≤ <in2> -->
<in3> op1 <in1> : <in2> co

5.35 <in1> ≤ <in3> ∧ <in3> ≤ <in2> -->
<in3>,<subexplist1> op1 <in1> : <in2> ,<bplist1> is
<subexplist1> op1 <bplist1 > co

5.36 <in> in <inlist> co
5.37 <in>,<inlist> in <inlist> co

5.38 <subexp1>,<subexplist1> is
va (<subexp1>) , va (<subexplist1>) co

5.39 <inlist1> is ⊀ <inlist1> ⊁ co

5.40 <bcs1>< dbcs > -->
<id1> [<subexplist1>] is <id1><bcs1> [va (<subexplist1>)] co

5.41 <id>bc [<subexplist>] is o co

5.42 <id1><bcs1><bc> [<subexplist1>] is <id1><bcs1> [<subexplist1>] co

5.43 <id1><bcs1> op2 (<id2> / <bcs2> < dbcs>) -->
<id1><bcs1>[<subexplist1>] is <id2><bcs2> [<subexplist1>] co

5.44 <type> array <id1><bcs1> [<bplist>] -->
<id1><bcs1> [< subexplist>] is o co

```
6.1    <dexp>  in  <dexplist> co
6.2    <decl dexp>  in  <decl dexplist> co

6.3    <dexp> , < dexplist>  in  <dexplist>  co
6.4    <decl dexp> , <decl dexplist>  in  <decl dexplist>  co

6.5    switch <id> := <dexplist>   in  <switchdeclaration> co

6.6    <bcs1><fas1> : switch <id1> := <dexplist1> ;
                  <decllist1> <stlist1>  end  is
       (  switch <id1><bcs1> co ≮  <bcs1><fas1> is
        ( <dexplist1>  in  <decl dexplist> co ≮  <bcs1><fas1>  is
         (  ≮  t <bcs1><fas1> is  (  switch <id1> := <dexplist1>co
         t <bcs1><fas1>a ) ≯ co <bcs1><fas1>a ) ≯ co <bcs1><fas1> a ) ≯ co
       <bcs1><fas1>a : <decllist1> <stlist1> end  )  co

6.7    switch <id1><bcs1>  is  ≮  switch <id1><bcs1> ≯ co

6.8    <specifier> <id1><bcs1> -->  switch <id1><bcs1>  is  o co

6.9    <bcs1><dbcs1> -->  switch <id1> := <dexplist1>  is
       ( ≮ switch <id1><bcs1><dbcs1> ≯ co <id1><bcs1> op4 <dexplist1> ) co

6.10   <id1><bcs1> op4 <dexpl>  is
       (  ≮  <id1><bcs1> [ <subexp>]  is  undefinedswitchdesignator ≯ co
          <  <id1> <bcs1> [ 1 ]  is  <dexpl>    ≯  )  co

6.11   <id1><bcs1> op4 <dexpl> , <dexplist1>  is
       (  ≮  <id1><bcs1>[ <subexp>] is undefinedswitchdesignator ≯ co
         ≮  <id1><bcs1> [ 1 ] is <dexpl>  ≯ co
         <id1><bcs1> [ 2 ]  op4 <dexplist1> )  co

6.12   <id1><bcs1>  [ <ui1> ] op4 <dexpl> , <dexplist1> is
       (   ≮ <id1><bcs1> [ <ui1> ] is <dexpl>  ≯ co
         <id1><bcs1>[ va  (  <ui1> + 1 ) ] op4   <dexplist1>  )  co

6.13   <id1><bcs1> [ <ui1> ]   op4  <dexpl>  is
       ≮  <id1><bcs1> [ <ui1> ]   is <dexpl>    ≯ co
```

```
7.1    <type>                  in  <valuespecifier> co
7.2    <type> array            in  <valuespecifier> co
7.3    <type> procedure        in  <valuespecifier> co

7.4    label                   in  <specifier> co
7.5    switch                  in  <specifier> co
7.6    procedure               in  <specifier> co
7.7    <valuespecifier>        in  <specifier> co

7.8    value  <idlist>;        in  <valuepart> co

7.9    <specifier><idlist> ;                in  <specpart> co
7.10   <specifier><idlist>;<specpart>       in  <specpart> co

7.11   <type>  procedure <id> ; <st>        in  <proceduredeclaration> co

7.12   <type>  procedure <id> ( <idlist> ) ;
       <valuepart> <specpart> <st> in  <proceduredeclaration> co

7.13   <id>   in  <procid> co

7.14   <procid> ( <actpalist> )  in  <procst> co

7.15   <procid>  in  <procst> co
7.16   <decl procid>          in  <decl procst> co

7.17   <bcs1> -->
       <id1> in <decl procid> is <id1><bcs1> in <decl procid> co

7.18   <id> bc  in  <decl procid>  is  false co

7.19   <id1><bcs1><bc>  in  <decl procid> is
       <id1><bcs1>      in  <decl procid> co

7.20   formal  <id1><bcs1> -->  <id1><bcs1>  in  <decl procid> co

7.21   <type> procedure  <id1><bcs1>  -->
       <id1><bcs1>  in <decl procid> co

7.22   <bcs1> -->  <id1> (  <decl actpalist1> )  in  <decl procst>  is
       <id1><bcs1> ( <decl actpalist1> )  in  <decl procst> co

7.23   <id> bc ( <actpalist> )  in  <decl procst>  is  false  co

7.24   <id1><bcs1><bc>   ( <actpalist1> )  in  <decl procst>  is
       <id1><bcs1>       ( <actpalist1> )  in  <decl procst>  co

7.25   formal  <id1><bcs1> -->
       <id1><bcs1> ( <actpalist1> )  in  <decl procst> co

7.26   <type> procedure <id1><bcs1>( <idlist1>) -->
       <id1><bcs1> ( <actpalist1>) in <decl procst>    is
       <idlist1>  op7  <actpalist1>  co
```

7.27 <id> op7 <actpa> co

7.28 <id> , <idlist1> op7 <actpa> , <actpalist1> is
<idlist1> op7 <actpalist1> co

7.29 <bcs1><fas1> : <type1> procedure <id1>; < st1> ;
<decllist1> <stlist1> end is
(<type1> procedure <id1><bcs1> co { <bcs1><fas1> is
begin integer dummy ; <st1> end in <declblock> co
firstlabelofprocedurebody <bcs1><fas1> co { <bcs1><fas1> is
({ t<bcs1><fas1> is (<type1> procedure <id1> op6
firstlabelofprocedurebody <bcs1><fas1> co t <bcs1><fas1> a) } co
<bcs1><fas1>a) } co <bcs1><fas1>a) } co
<bcs1><fas1>a : <decllist1> <stlist1> end) co

7.30 <bcs1><fas1> : <type1> procedure <id1>(<idlist1>) ;<valuepart1>
<specpart1> <st1> ; <decllist1> <stlist1> end is
(<type1> procedure <id1><bcs1>(<idlist1>)co { <bcs1><fas1> is
(begin co formal <idlist1> co begin integer dummy ; <st1> end
in <declblock> co firstlabelofprocedurebody <bcs1><fas1> co end co
{ <bcs1><fas1> is ({ t <bcs1><fas1> is
(<type1> procedure <id1> (<idlist1>); <valuepart1><specpart1>
op6 firstlabelofprocedurebody <bcs1><fas1> co t <bcs1><fas1>a) } co
<bcs1><fas1> a) } co <bcs1><fas1> a) } co
<bcs1><fas1> a : <decllist1> <stlist1> end) co

7.31 formal <id1> , <idlist1> is (formal <id1> co formal <idlist1>) co

7.32 <bcs1> --> formal <id1> is { formal <id1><bcs1> } co

7.33 <type1> procedure <id1><bcs1> is
{ <type1> procedure <id1><bcs1> } co

7.34 <specifier> <id1><bcs1> --> <type> procedure <id1><bcs1> is o co

7.35 <type1> procedure <id1><bcs1> (<idlist1>) is
{ <type1> procedure <id1><bcs1> (<idlist1>) } co

7.36 <specifier> <id1><bcs1> -->
<type> procedure <id1><bcs1> (<idlist>) is o co

7.37 <bcs1><bc1> --> firstlabelofprocedurebody <bcs1><fas1> is
{ firstlabelofprocedurebody <bcs1><fas1> is < <bcs1><bc1> a } } co

7.38 <bcs1><dbcs> --> <type1> procedure <id1> op6
firstlabelofprocedurebody <bcs2><fas1> is
<type1> procedure <id1><bcs1> op16
va (firstlabelofprocedurebody <bcs2><fas1>) co

7.39 <type> procedure <id1><bcs1> op16 <bcs2>a is
{ <type1> procedure <id1><bcs1> is <bcs2>a } co

7.40 <bcs1><dbcs> --> <type1> procedure <id1> (< idlist1>);
<valuepart1><specpart1> op6 firstlabelofprocedurebody <bcs2><fas1> is
⌈ <type1> procedure <id1><bcs1> op16
va (firstlabelofprocedurebody <bcs2><fas1>) co
<type1> procedure <id1><bcs1>(< idlist1>) ;
<valuepart1> <specpart1> ⌉ co

7.41 <valuespecifier> <id>, in < leftformal> co

7.42 <leftformal> in <leftformallist> co
7.43 <leftformal><leftformallist> in <leftformallist> co

7.44 , <valuespecifier><id> in <rightformal> co

7.45 <rightformal> in <rightformallist> co
7.46 <rightformal><rightformallist> in <rightformallist> co

7.47 < leftformallist> <valuespecifier> <id><rightformallist>
in <extformallist> co

7.48 <type1> procedure <id1><bcs1> (<extformallist1>) ; <specpart> is
ǂ <type1> procedure <id1><bcs1> (<extformallist1>) ǂ co

7.49 (<type1> procedure <id1> <bcs1>
⌈ <leftformallist1><id2><rightformallist1>); value <id2>, <idlist1>;
<specpart1> <valuespecifier1><leftformallist2>
<id2><rightformallist2> ; <specpart2>) is
<type1> procedure<id1><bcs1>
⌈ <leftformallist1> <valuespecifier1><id2><rightformallist1>);
value <idlist1> ; <specpart1> <valuespecifier1><leftformallist2>
<id2><rightformallist2> ; <specpart2> co

7.50 <type1> procedure <id1><bcs1>
⌈ <leftformallist1> <id2><rightformallist1>) ;
value <id2> ; <specpart> <valuespecifier1> <leftformallist>
<id2><rightformallist> ; <specpart> is
<type1> procedure <id1><bcs1>
⌈ <leftformallist1> <valuespecifier1><id2><rightformallist1>) ; co

```
8.1    <intvar> :=          in  <intleftpart> co
8.2    <intprocid> :=       in  <intleftpart> co

8.3    <decl intvar> :=             in <decl intleftpart> co
8.4    <decl intprocid> :=          in <decl intleftpart> co

8.5    <booleanvar> :=              in  <booleanleftpart> co
8.6    <booleanprocid> :=           in  <booleanleftpart> co

8.7    <decl booleanvar> :=         in  <decl booleanleftpart> co
8.8    <decl booleanprocid>:=       in  <decl booleanleftpart> co

8.9    <intleftpart>                            in  <intleftpartlist> co
8.10   <intleftpart><intleftpartlist>           in  <intleftpartlist> co

8.11   <decl intleftpart>                               in <decl intleftpartlist> co
8.12   <decl intleftpart><decl intleftpartlist>         in <decl intleftpartlist> co

8.13   <booleanleftpart>                                in  <booleanleftpartlist> co
8.14   <booleanleftpart><booleanleftpartlist>           in  <booleanleftpartlist> co

8.15   <decl booleanleftpart>      in  <decl booleanleftpartlist> co
8.16   <decl booleanleftpart><decl booleanleftpartlist>
       in  <decl booleanleftpartlist> co

8.17   <intleftpartlist><aexp>                  in  <assst> co
8.18   <booleanleftpartlist><bexp>              in  <assst> co

8.19   <decl intleftpartlist><decl aexp>                in  <decl assst> co
8.20   <decl booleanleftpartlist><decl bexp>            in  <decl assst> co

8.21   <type><id><bcs> :=                               in  <extleftpart> co
8.22   <type> array <id><bcs>[ <subexplist>]:=          in  <extleftpart> co

8.23   <intleftpart>          in  <extleftpartlist> co
8.24   <booleanleftpart>      in  <extleftpartlist> co

8.25   <extleftpart>                            in  <extleftpartlist> co
8.26   <extleftpart><extleftpartlist>           in  <extleftpartlist> co

8.27   <extleftpart1><extleftpartlist1><exp1>     is
       <extleftpart1><extleftpartlist1> va (  <exp1> )  co

8.28   <in>                   in  <constant> co
8.29   <logicalvalue>         in  <constant> co

8.30   <extleftpart1><extleftpartlist1><constant1>   is
       ( <extleftpart1><constant1> co<extleftpartlist1><constant1> ) co

8.31   <extleftpart><constant>   is  o  co
```

8.32 <bcs1><dbcs> --> <id1>:= <extleftpartlist1><exp1> is
<id1><bcs1> := <extleftpartlist1><exp1> co

8.33 <id>bc := <extleftpartlist><exp> is o co

8.34 <id1><bcs1><bc> := <extleftpartlist1><exp1> is
<id1><bcs1> := <extleftpartlist1><exp1> co

8.35 <id1><bcs1> op2 (<id2> / <bcs2><dbcs>) -->
<id1><bcs1>:= <extleftpartlist1> <exp1> is
<id2><bcs2>:= < extleftpartlist1><exp1> co

8.36 <id1><bcs1> op2 (<id2>[<subexplist1>] / <bcs2><dbcs>)-->
<id1><bcs1> := <extleftpartlist1> <exp1> is
<id2><bcs2>[va (<subexplist1>)] := <extleftpartlist1> <exp1> co

8.37 <type1><id1><bcs1> -->
<id1><bcs1>:= <extleftpartlist1><exp1> is
<extleftpartlist1> <type1><id1><bcs1>:= <exp1> co

8.38 <bcs1><dbcs1> --> <id1>[<subexplist1>]:=<extleftpartlist1><exp1>is
<id1><bcs1>[va (<subexplist1>)]:= <extleftpartlist1><exp1> co

8.39 <id>bc [<subexplist>] := <extleftpartlist> <exp> is o co

8.40 <id1><bcs1><bc>[<subexplist1>]:= <extleftpartlist1><exp1> is
<id1><bcs1>[<subexplist1>] := <extleftpartlist1> <exp1> co

8.41 <id1><bcs1> op2 (<id2> / <bcs2><dbcs>) -->
<id1><bcs1>[<subexplist1>]:= <extleftpartlist1><exp1> is
<id2><bcs2> [<subexplist1>] := <extleftpartlist1><exp1> co

8.42 <type1> array <id1><bcs1> [<intbplist1>] -->
<id1><bcs1>[<subexplist1>] := <extleftpartlist1><exp1> is
(<subexplist1> op1 <intbplist1> co
<extleftpartlist1><type1>array<id1><bcs1>[<subexplist1>]:=<exp1>) co

8.43 integer <id1><bcs1> := <in1> is ⊀ <id1><bcs1> is <in1> ⊁ co

8.44 boolean <id1><bcs1> := <logicalvalue1> is
⊀ <id1><bcs1> is <logicalvalue1> ⊁ co

8.45 integer array <id1><bcs1> [<subexplist1>] := <in1> is
⊀ <id1><bcs1> [<subexplist1>] is <in1> ⊁ co

8.46 boolean array <id1><bcs1>[<subexplist1>] := <logicalvalue1> is
⊀ <id1><bcs1> [<subexplist1>] is <logicalvalue1> ⊁ co

```
9.1    goto ( <dexpl> ) is goto <dexpl> co

9.2    goto if <bexpl> then <sdexpl> else <dexpl> is
       goto if va ( <bexpl> ) then <sdexpl> else <dexpl> co

9.3    goto if true then <sdexpl> else <dexp> is goto <sdexpl> co

9.4    goto if false then <sdexp> else <dexpl> is goto <dexpl> co

9.5    if <bexpl> then goto <dexpl> is
       if va ( <bexpl> ) then goto <dexpl> co

9.6    if true then goto <dexpl> is goto <dexpl> co

9.7    if false then goto <dexp> co

9.8    <bcs1><dbcs> --> goto <label1> is goto <label1><bcs1> co
9.9    goto <label>bc is o co
9.10   goto <label1><bcs1><bc> is goto <label1><bcs1> co

9.11   <id1><bcs1> op2 ( <dexpl> / <bcs2><dbcs1> ) -->
       goto <id1><bcs1> is
       ( savebn <id1><bcs1> co ⊀ <bcs2><dbcs1> ⊁ co goto <dexpl> co
       resetbn <id1><bcs1>) co

9.12   ( label <label1><bcs1><dbcs1> is t <bcs2><fas1> ) -->
       goto <label1><bcs1> is ( ⊀ <bcs1><dbcs1> ⊁ co t<bcs2><fas1> ) co

9.13   <bcs1><dbcs> --> goto <id1>[ <subexpl> ] is
                        goto <id1><bcs1> [ va ( <subexpl> ) ] co
9.14   goto <id>bc [ <subexp> ] is o co
9.15   goto <id1><bcs1><bc>[ <subexpl> ] is goto <id1><bcs1>[<subexpl>] co

9.16   <id1><bcs1> op2 ( <id2>/ <bcs2><dbcs> ) -->
       goto <id1><bcs1>[<subexpl>] is goto <id2><bcs2>[ <subexpl>] co

9.17   switch <id1><bcs1><dbcs1> --> goto <id1><bcs1>[<subexpl>] is
       ( savebn <id1><bcs1> co ⊀ <bcs1><dbcs1> ⊁ co
       op13 <id1><bcs1>[ <subexpl>] co resetbn <id1><bcs1> co
       goto va ( <id1><bcs1>[<subexpl>] ) ) co

9.18   ( <id1><bcs1>[<subexpl>] is <dexpl> ) -->
       op13 <id1><bcs1>[<subexpl>] is
       <id1><bcs1>[<subexpl>] op14 <dexpl> co

9.19   ( <id1><bcs1>[<subexpl>] is undefinedswitchdesignator ) -->
       op13 <id1><bcs1>[<subexpl>] is
       ⊀ <id1><bcs1>[<subexpl>] is ⊀ undefinedswitchdesignator ⊁ ⊁ co
```

```
9.20   <id1><bcs1>[<subexpl>] op14 ( <dexpl> ) is
       <id1><bcs1>[<subexpl>] op14 <dexpl> co

9.21   <id1><bcs1>[<subexpl>] op14
        if <bexpl> then <sdexpl> else <dexpl> is
       <id1><bcs1>[<subexpl>]  op14
       if  va ( <bexpl> )  then <sdexpl> else <dexpl> co

9.22   <id1><bcs1>[<subexpl>] op14
       if  true  then <sdexpl> else <dexp>  is
       <id1><bcs1>[<subexpl>] op14  <sdexpl> co

9.23   <id1><bcs1>[<subexpl>] op14
       if  false  then <sdexp>  else  <dexpl>  is
       <id1><bcs1>[ <subexpl>] op14 <dexpl> co

9.24   <id1><bcs1>[<subexpl>] op14 <label1> is
       ⊀  <id1><bcs1>[<subexpl>] is ⊀  <label1> ⊁ ⊁ co

9.25   <id1><bcs1>[<subexpl>] op14 <id2>[<subexp2>] is
       ⊀  <id1><bcs1>[<subexpl>] is ⊀ <id2>[<subexp2>] ⊁ ⊁ co

9.26   goto  undefinedswitchdesignator  co
```

```
10.1    <bcs1><dbcs>  -->  st : <id1>  is  st : <id1><bcs1> co

10.2    st :  <id>bc  is  o co

10.3    st :  <id1><bcs1><bc>  is  st :  <id1><bcs1> co

10.4    <id1><bcs1> op2  ( <id2>/ <bcs2><dbcs>  ) -->
        st :<id1><bcs1>  is  st  :  <id2><bcs2> co

10.5    (  procedure <id1><bcs1> is  <bcs2>a )  -->
        st : <id1><bcs1> is
        (  enterprocedure <bcs1> co <bcs2>a co exitprocedure )  co

10.6    <bcs1><dbcs> -->
        st :  <id1>(<actpalist1>) is st : <id1><bcs1>(<actpalist1>) co

10.7    st : <id>bc ( <actpalist> ) is o co

10.8    st : <id1><bcs1><bc> ( <actpalist1>) is
        st : <id1><bcs1>      ( <actpalist1>) co

10.9    <id1><bcs1> op2 ( <id2> / <bcs2><dbcs> )  -->
        st : <id1><bcs1> ( <actpalist1> ) is
        st : <id2><bcs2> ( <actpalist1> ) co

10.10   procedure <id1><bcs1> ( <extformallist1>) -->
        st : <id1><bcs1> ( <actpalist1> ) is
        (  enterprocedure <bcs1> co
        <id1><bcs1> (<extformallist1>) op8 <id1><bcs1> ( <actpalist1> ) co
        exitprocedure )  co

10.11   <bcs1><dbcs>  -->  <id1> ( <actpalist1>)  is
                           <id1><bcs1> ( <actpalist1> ) co

10.12   <id> bc ( <actpalist>)  is o  co

10.13   <id1><bcs1><bc> ( <actpalist1> ) is
        <id1><bcs1>      ( <actpalist1> ) co

10.14   <id1><bcs1>  op2  (  <id2> / <bcs2><dbcs> ) -->
        <id1><bcs1> ( <actpalist1> )  is  <id2><bcs2> ( <actpalist1> ) co

10.15   <type1> procedure <id1><bcs1> ( <extformallist1> ) -->
        <id1><bcs1> ( <actpalist1> ) is
        (  enterprocedure <bcs1> co begin co <type1><id1> co
        <id1><bcs1>( <extformallist1>) op8 <id1><bcs1> ( <actpalist1>) co
        functionvalue op15  <id1> co end co exitprocedure co
        functionvalue )  co
```

10.16 <bcs1><dbcs1> --> enterprocedure <bcs2> is
≮ <bcs2> d <bcs1><dbcs1> ≯ co

10.17 <bcs1>d <bcs2><dbcs1> --> exitprocedure is < <bcs2><dbcs1> ≯ co

10.18 <id1><bcs1> (<type1><id2> , <extformallist1>) op8
<id1><bcs1> (<actpa1> , <actpalist1>) is
(begin co <type1><id2> co <type1><id2> op9 <actpa1> co
<id1><bcs1>(<extformallist1>) op8 <id1><bcs1>(<actpalist1>) co end) co

10.19 (<type> procedure <id1><bcs1> is <bcs2>a) - >
<id1><bcs1>(<type1><id2>) op8 <id1><bcs1>(<actpa1>) is
(begin co <type1><id2> co <type1><id2> op9 <actpa1> co
<bcs2>a co end) co

10.20 <id1><bcs1> (<id2> , <extformallist1>) op8
<id1><bcs1> (<actpa1> , <actpalist1>) is
<id1><bcs1> (<extformallist1> , <id2>) op8
<id1><bcs1> (<actpalist1> , <actpa1>) co

10.21 <id1><bcs1> (<id2> , <idlist2>) op8
<id1><bcs1> (<actpa1> , <actpalist1>) is
(begin co <id2> op10 <actpa1> co
<id1><bcs1> (<idlist2>) op8 <id1><bcs1> (<actpalist1>) co end) co

10.22 (<type> procedure <id1><bcs1> is <bcs2> a) -->
<id1><bcs1> (<id2>) op8 <id1><bcs1> (<actpa1>) is
(begin co <id2> op10 <actpa1> co <bcs2> a co end) co

10.23 <bcs1> d <bcs2> <dbcs1> --> <type1><id1> op9 <actpa1> is
(≮ <bcs2><dbcs1> ≯ co <type1><id1><bcs1> op17 <actpa1> co
≮ <bcs1> d <bcs2> <dbcs1> ≯) co

10.24 <type1><id1><bcs1> op17 <actpa1> is
<type1><id1><bcs1> op17 va (<actpa1>) co

10.25 <type><id><bcs> op17 <constant> is o co

10.26 integer <id1><bcs1> op17 <in1> is ≮ <id1><bcs1> is <in1> ≯ co

10.27 boolean <id1><bcs1> op17 <logicalvalue1> is
≮ <id1><bcs1> is <logicalvalue1> ≯ co

10.28 <bcs1> d <bcs2> <dbcs1> -->
<id1> op10 <actpa1> is

≮ <id1><bcs1> op2 (<actpa1> / <bcs2><dbcs1>) ≯ co

```
10.29   <id1><bcs1> ( <type1> procedure  <id2> , <extformallist1> ) op8
        <id1><bcs1> ( <actpa1> , <actpalist1> )
        is
        <id1><bcs1> ( <type1><id2> , <extformallist1> ) op8
        <id1><bcs1> ( <actpa1> , <actpalist1> )  co

10.30   <id1><bcs1> ( <type1> procedure <id2> ) op8
        <id1><bcs1> ( <actpa1> )
        is
        <id1><bcs1>( <type1><id2> ) op8
        <id1><bcs1> ( <actpa1> ) co

10.31   <id1><bcs1>( <type1>array <id2> , <extformallist1> )  op8
        <id1><bcs1>  ( <id3> , <actpalist1> )
        is
        (  begin co <type1> array <id2>  op12 <id3> co
        <id1><bcs1> ( <extformallist1>) op8 <id1><bcs1> ( <actpalist1> ) co
        end )  co

10.32   (  <type> procedure <id1><bcs1> is <bcs2> a )  -->
        <id1><bcs1> ( <type1> array <id2> ) op8   <id1><bcs1> ( <id3> )
        is
        (   begin co <type1> array <id2> op12 <id3> co <bcs2> a co end )  co

10.33   <bcs1> d <bcs2> <dbcs> -->  <type1> array <id1> op12  <id2> is
        <type1> array <id1><bcs1> op12  <id2><bcs2> co

10.34   <type> array <id>  <bcs> op12 <id> bc  is  o  co

10.35   <type1> array <id1><bcs1>   op12 <id2><bcs2><bc>  is
        <type1> array <id1><bcs1>   op12  <id2><bcs2> co

10.36   <id2><bcs2>  op2 ( <id3> / <bcs3> <dbcs> )  -->
        <type1> array <id1><bcs1> op12 <id2><bcs2> is
        <type1> array <id1><bcs1> op12 <id3><bcs3> co

10.37   <type1>array <id2><bcs2>[ <intbplist1> ] -->
        <type1>array <id1><bcs1>  op12 <id2><bcs2>  is
        (  ≮  <type1> array <id1><bcs1> [ <intbplist1> ]  ≯ co
        <id1><bcs1>[ <intbplist1> ]  op11  <id2><bcs2>  )  co
```

```
10.38   <in> , in <leftintlist> co

10.39   <leftintlist><leftintlist>   in  <leftintlist> co

10.40   <id1><bcs1> [ <leftintlist1>  <in1> : <in2> ]  op11 <id2><bcs2>
        is
        ( <id1><bcs1>[ < leftintlist1> <in1> ]  :=
        <id2><bcs2>[ <leftintlist1><in1> ] co
        <id1><bcs1>[ <leftintlist1> va ( <in1> + 1 )  :  <in2> ] op11
        <id2><bcs2> )  co

10.41   <id1><bcs1>[<leftintlist1><in1>:<in2> , <bplist1>] op11 <id2><bcs2>
        is
        ( <id1><bcs1>[<leftintlist1><in1> , <bplist1>] op11 <id2><bcs2> co
         <id1><bcs1>[<leftintlist1> va ( <in1>+1) : <in2> , <bplist1>] op11
        <id2><bcs2> )  co

10.42   <in1>  =  <in2>  -->
        <id1><bcs1>[<leftintlist1><in1>:<in2> , <bplist1>] op11 <id2><bcs2>
        is
        <id1><bcs1>[ <leftintlist1><in1> , <bplist1>] op11 <id2><bcs2>   co

10.43   <in1> = <in2> -->
        <id1><bcs1> [ <leftintlist1> <in1> : <in2> ] op11 <id2><bcs2> is
        <id1><bcs1>[<leftintlist1><in1>] :=<id2><bcs2>[<leftintlist1><in1>] co

10.44   ( <type1> procedure <id1><bcs1> is <bcs2>a  ) -->
        st : <id1><bcs1>  is
        ( enterprocedure <bcs1> co begin co <type1><id1> co <bcs2> a co
        end co exitprocedure )  co

10.45.   <type1> procedure <id1><bcs1>(<extformallist1>) -->
        st : <id1><bcs1> (<actpalist1>) is
        ( enterprocedure <bcs1> co begin co <type1><id1> co
        <id1><bcs1>(<extformallist1>) op8 <id1><bcs1>(<actpalist1>) co
        end co exitprocedure )  co
```

```
11.1    <aexp>                                        in  <forlistel> co
11.2    <aexp>  while  <bexp>                         in  <forlistel> co
11.3    <aexp>  step  <aexp>  until  <aexp>           in  <forlistel> co

11.4    <forlistel>                      in  <forlist> co
11.5    <forlistel> , <forlist>          in  <forlist> co

11.6    for  <intvar> := <forlist>  do  <st>          in  <forst> co
11.7    <label> : <forst>                             in  <forst> co

11.8    <bcs1><fas1> : for <intvar1> :=<forlistel1> , <forlist1> do <st1>
                          <specialstlist1> <end1>
        is
        <bcs1><fas1> : for1 <intvar1> := <forlistel1> do <st1> ;
        for <intvar1> := <forlist1> do <st1> <specialstlist1> <end1> co

11.9    <bcs1><fas1> : for <intvar1> := <forlistel1>  do <st1>
                          <specialstlist1> <end1>
        is
        ( ≮   <bcs1><fas1> is  ( ≮ <bcs1><fas1> is  (  begin co
        ≮ t<bcs1><fas1>  is  t <bcs1><fas1>fa ≯ co
        <bcs1><fas1>fa ) ≯ co <bcs1><fas1>fa ) ≯ co
        <bcs1><fas1>fa : forbegin <intvar1> := <forlistel1>;
                             <st1> forend <intvar1> co
        <bcs1><fas1>a : <specialstlist1> <end1> )  co

11.10   <bcs1><fas1> : for1 <intvar1> := <forlistel1> do <st1>
                          <specialstlist1> <end1>
        is
        ( ≮ <bcs1><fas1> is  ( ≮ <bcs1><fas1> is (  begin co
        ≮  t <bcs1><fas1> is t <bcs1><fas1>fa ≯ co
        <bcs1><fas1>fa ) ≯ co <bcs1><fas1>fa ) ≯ co
        <bcs1><fas1>fa : forbegin <intvar1> := <forlistel1>;
                          <st1> forend co
        <bcs1><fas1>a : <specialstlist1> <end1> )  co
```

11.11 <bcs1><fas1>f <as2> : forend is
⊀ <bcs1><fas1>f <as2> is (⊀ <bcs1><fas1>f <as2> is
(⊀ t<bcs1><fas1>f<as2> is (end co t<bcs1><fas1>a) ⊁ co
<bcs1><fas1>a) ⊁ co <bcs1><fas1>a) ⊁ co

11.12 <bcs1><fas1>f <as2> : forend <intvar1> is
⊀ <bcs1><fas1>f <as2> is (⊀ <bcs1><fas1>f <as2> is
(⊀ t<bcs1><fas1>f <as2> is (end co ⊀ <intvar1> is o ⊁ co
t<bcs1><fas1>a) ⊁ co <bcs1><fas1>a) ⊁ co <bcs1><fas1>a) ⊁ co

11.13 <bcs1><fas1> : forbegin <intvar1> := <aexp1> ; <st1>
forend <intvar1>
is
<bcs1><fas1> : <intvar1> := <aexp1>; < st1> forend <intvar1> co

11.14 <bcs1><fas1> : forbegin <intvar1> := <aexp1> while <bexp1>;
< st1> forend <intvar1>
is
<bcs1><fas1> : <bcs1><fas1> l : <intvar1> := <aexp1>;
if <bexp1> then begin <st1> ; goto <bcs1><fas1> l end
forend < intvar1> co

11.15 <bcs1><fas1> : forbegin <intvar1> := <aexp1> step <aexp2>
until <aexp3>; <st1> forend <intvar1>
is
<bcs1><fas1> : <intvar1> := <aexp1> ; <bcs1><fas1>l :
begin integer procedure sign (f); value f; integer f;
sign:= if f > 0 then 1 else if f = 0 then 0 else - 1;
if (<intvar1> - <aexp3>) × sign (<aexp2>) > 0
then goto <bcs1><fas1>m
end;
<st1> ; <intvar1> := <intvar1> + <aexp2>; goto <bcs1><fas1>l;
<bcs1><fas1>m : forend <intvar1> co

```
12.1    <aexp>  in  <exp> co
12.2    <bexp>  in  <exp> co
12.3    <dexp>  in  <exp> co

12.4    <decl aexp>    in  <decl exp> co
12.5    <decl bexp>    in  <decl exp> co
12.6    <decl dexp>    in  <decl exp> co

12.7    <id>    in  <intvarid> co
12.8    <id>    in  <booleanvarid> co
12.9    <id>    in  <intarrayid> co
12.10   <id>    in  <booleanarrayid> co

12.11   <bcs1> --> <id1> in <decl intvarid> is
                   <id1><bcs1> in <decl intvarid> co
12.12   <bcs1> --> <id1> in <decl booleanvarid> is
                   <id1><bcs1> in <decl booleanvarid> co
12.13   <bcs1> --> <id1> in <decl intarrayid> is
                   <id1><bcs1> in <decl intarrayid> co
12.14   <bcs1> --> <id1> in <decl booleanarrayid> is
                   <id1><bcs1> in <decl booleanarrayid> co

12.15   <id>bc  in <decl intvarid>is false co
12.16   <id>bc  in <decl booleanvarid> is false co
12.17   <id>bc  in <decl intarrayid> is false co
12.18   <id>bc  in <decl booleanarrayid> is false co

12.19   <id1><bcs1><bc> in <decl intvarid> is
        <id1><bcs1>     in <decl intvarid> co
12.20   <id1><bcs1><bc> in <decl booleanvarid> is
        <id1><bcs1>     in <decl booleanvarid> co
12.21   <id1><bcs1><bc> in <decl intarrayid> is
        <id1><bcs1>     in <decl intarrayid> co
12.22   <id1><bcs1><bc> in <decl booleanarrayid> is
        <id1><bcs1>     in <decl booleanarrayid> co

12.23   formal <id1><bcs1> --> <id1><bcs1> in <decl intvarid> co
12.24   formal <id1><bcs1> --> <id1><bcs1> in <decl booleanvarid> co
12.25   formal <id1><bcs1> --> <id1><bcs1> in <decl intarrayid> co
12.26   formal <id1><bcs1> --> <id1><bcs1> in <decl booleanarrayid> co

12.27   integer <id1><bcs1>      --> <id1><bcs1> in <decl intvarid> co
12.28   boolean <id1><bcs1>      --> <id1><bcs1> in <decl booleanvarid> co
12.29   integer array <id1><bcs1> -->
        <id1><bcs1> in <decl intarrayid> co
12.30   boolean array <id1><bcs1>-->
        <id1><bcs1> in <decl booleanarrayid> co
```

12.31 <intvarid> in <intvar> co
12.32 <booleanvarid> in <booleanvar> co

12.33 <decl intvarid> in <decl intvar> co
12.34 <decl booleanvarid> in <decl booleanvar> co

12.35 <aexp> in <subexp> co

12.36 <subexp> in <subexplist> co
12.37 <subexp> , <subexplist> in <subexplist> co

12.38 <decl aexp> in <decl subexp> co

12.39 <decl subexp> in <decl subexplist> co
12.40 <decl subexp> , <decl subexplist> in <decl subexplist> co

12.41 <intarrayid>[<subexplist>] in <intvar> co
12.42 <booleanarrayid>[<subexplist>] in <booleanvarid> co

12.43 <decl intarrayid>[<decl subexplist>] in <decl intvar> co

12.44 <decl booleanarrayid>[<decl subexplist>]
in <decl booleanvar> co

12.45 <id> in <intprocid> co
12.46 <id> in <booleanprocid> co

12.47 <bcs1> --> <id1> in <decl intprocid> is
<id1><bcs1> in <decl intprocid> co
12.48 <bcs1> --> <id1> in <decl booleanprocid> is
<id1><bcs1> in <decl booleanprocid> co

12.49 <id>bc in <decl intprocid> is false co
12.50 <id>bc in <decl booleanprocid> is false co

12.51 <id1><bcs1><bc> in <decl intprocid> is
<id1><bcs1> in <decl intprocid> co
12.52 <id1><bcs1><bc> in <decl booleanprocid> is
<id1><bcs1> in <decl booleanprocid> co

12.53 formal <id1><bcs1> --> <id1><bcs1> in <decl intprocid> co
12.54 formal <id1><bcs1> --> <id1><bcs1> in <decl booleanprocid> co

12.55 integer procedure <id1><bcs1> -->
<id1><bcs1> in <decl intprocid> co

12.56 boolean procedure <id1><bcs1> -->
<id1><bcs1> in <decl booleanprocid> co

12.57 <intprocid> in <intfunctdes> co
12.58 <booleanprocid> in <booleanfunctdes> co

12.59 <decl intprocid> in <decl intfunctdes> co
12.60 <decl booleanprocid> in <decl booleanfunctdes> co

```
12.61   <exp>               in  <actpa> co
12.62   <intarrayid>        in  <actpa> co
12.63   <booleanarrayid>    in  <actpa> co
12.64   <switchid>          in  <actpa> co
12.65   <intprocid>         in  <actpa> co
12.66   <booleanprocid>     in  <actpa> co
12.67   <procid>            in  <actpa> co

12.68   <decl exp>                in  <decl actpa> co
12.69   <decl intarrayid>         in  <decl actpa> co
12.70   <decl booleanarrayid>     in  <decl actpa> co
12.71   <decl switchid>           in  <decl actpa> co
12.72   <decl intprocid>          in  <decl actpa> co
12.73   <decl booleanprocid>      in  <decl actpa> co
12.74   <decl procid>             in  <decl actpa> co

12.75   <actpa>             in  <actpalist> co

12.76   <actpa> , <actpalist>     in  <actpalist>  co

12.77   <decl actpa>                         in  <decl actpalist> co
12.78   <decl actpa> , <decl actpalist>      in  <decl actpalist> co

12.79   <id>(<actpalist>)         in  <intfunctdes>       co
12.80   <id>(<actpalist>)         in  <booleanfunctdes>   co

12.81   <bcs1> --> <id1>(<decl actpalist1>)       in  <decl intfunctdes> is
                   <id1><bcs1>(<decl actpalist1>)  in  <decl intfunctdes> co

12.82   <bcs1> --> <id1>(<decl actpalist1>) in  <decl booleanfunctdes> is
             <id1><bcs1>(<decl actpalist1>) in  <decl booleanfunctdes> co

12.83   <id>bc(<actpalist>)       in  <decl intfunctdes> is false co
12.84   <id>bc(<actpalist>)       in  <decl booleanfunctdes>  is false co

12.85   <id1><bcs1><bc>(<actpalist1>)      in  <decl intfunctdes> is
        <id1><bcs1>     (<actpalist1>)     in  <decl intfunctdes> co

12.86   <id1><bcs1><bc>(<actpalist1>)      in  <decl booleanfunctdes> is
        <id1><bcs1>     (<actpalist1>)     in  <decl booleanfunctdes> co

12.87   formal <id1><bcs1> --> <id1><bcs1>(<actpalist1>)
        in  <decl intfunctdes> co

12.88   formal <id1><bcs1> --> <id1><bcs1>(<actpalist1>)
        in  <decl booleanfunctdes> co

12.89   integer procedure <id1><bcs1>(<idlist1>) -->
        <id1><bcs1>(<actpalist1>)  in  <decl intfunctdes> is
        <idlist1>  op7  <actpalist1> co

12.90   boolean procedure <id1><bcs1>(<idlist1>) -->
        <id1><bcs1>(<actpalist1>)  in  <decl booleanfunctdes> is
        <idlist1>  op7  <actpalist1> co
```

13.1 + in <pm> co
13.2 - in <pm> co

13.3 × in <multop> co
13.4 : in <multop> co

13.5 <ui> in <primary> co
13.6 <intvar> in <primary> co
13.7 <intfunctdes> in <primary> co
13.8 (<aexp>) in <primary> co

13.9 <ui> in <decl primary> co
13.10 <decl intvar> in <decl primary> co
13.11 <decl intfunctdes> in <decl primary> co
13.12 (<decl aexp>) in <decl primary> co

13.13 <primary> in <factor> co
13.14 <factor> ↑ <primary> in <factor> co

13.15 <decl primary> in <decl factor> co
13.16 <decl factor> ↑ <decl primary> in <decl factor> co

13.17 <factor> in <term> co
13.18 <term><multop><factor> in <term> co

13.19 <decl factor> in <decl term> co
13.20 <decl term><multop><decl factor> in <decl term> co

13.21 <term> in <saexp> co
13.22 <pm><term> in <saexp> co
13.23 <saexp><pm><term> in <saexp> co

13.24 <decl term> in <decl saexp> co
13.25 <pm><decl term> in <decl saexp> co
13.26 <decl saexp><pm><decl term> in <decl saexp> co

13.27 <saexp> in <aexp> co

13.28 if <bexp> then <saexp> else <aexp> in <aexp> co

13.29 <decl saexp> in <decl aexp> co

13.30 if <decl bexp> then <decl saexp> else <decl aexp>
in <decl aexp> co

14.1	<	in <relop> co
14.2	>	in <relop> co
14.3	=	in <relop> co
14.4	≠	in <relop> co
14.5	≤	in <relop> co
14.6	≥	in <relop> co
14.7	<logicalvalue>	in <bprimary> co
14.8	<booleanvar>	in <bprimary> co
14.9	<saexp><relop><saexp>	in <bprimary> co
14.10	(<bexp>)	in <bprimary> co
14.11	<booleanfunctdes>	in <bprimary> co
14.12	<logicalvalue>	in <decl bprimary> co
14.13	<decl booleanvar>	in <decl bprimary> co
14.14	<decl saexp><relop><decl saexp>	in <decl bprimary> co
14.15	<decl booleanfunctdes>	in <decl bprimary> co
14.16	(<decl bexp>)	in <decl bprimary> co
14.17	<bprimary>	in <bsecondary> co
14.18	¬ <bprimary>	in <bsecondary> co
14.19	<bsecondary>	in <bfactor> co
14.20	<bfactor> ∧ <bsecondary>	in <bfactor> co
14.21	<bfactor>	in <bterm> co
14.22	<bterm> ∨ <bfactor>	in <bterm> co
14.23	<bterm>	in <implication> co
14.24	<implication> $\underline{\neg}$ <bterm>	in <implication> co
14.25	<implication>	in <sbexp> co
14.26	<sbexp> $\underline{=}$ <implication>	in <sbexp> co
14.27	<sbexp>	in <bexp> co
14.28	if <bexp> then <sbexp> else <bexp>	in <bexp> co
14.29	<decl bprimary>	in <decl bsecondary> co
14.30	¬ <decl bprimary>	in <decl bsecondary> co
14.31	<decl bsecondary>	in <decl bfactor> co
14.32	<decl bfactor> ∧ <decl bsecondary>	in <decl bfactor> co
14.33	<decl bfactor>	in <decl bterm> co
14.34	<decl bterm> ∨ <decl bfactor>	in <decl bterm> co
14.35	<decl bterm>	in <decl implication> co
14.36	<decl implication> $\underline{\neg}$ <decl bterm>	in <decl implication> co
14.37	<decl implication>	in <decl sbexp> co
14.38	<decl sbexp> $\underline{=}$ <decl implication>	in <decl sbexp> co
14.39	<decl sbexp>	in <decl bexp> co
14.40	if <decl bexp> then <decl sbexp> else <decl bexp>	in <decl bexp> co

```
15.1    <id>     in   <label> co
15.2    <ui>     in   <label> co
15.3    <id>     in   <switchid> co

15.4    <bcs1> -->      <label1>            in   <decl label> is
                        <label1><bcs1>      in   <decl label> co

15.5    <bcs1> -->      <id1>               in   <decl switchid> is
                        <id1><bcs1>         in   <decl switchid> co

15.6    <id>bc          in   <decl label>           is   false  co
15.7    <ui>bc          in   <decl label>           is   false  co
15.8    <id>bc          in   <decl switchid>        is   false  co

15.9    <label1><bcs1><bc>                  in   <decl label> is
        <label1><bcs1>                      in   <decl label> co

15.10   <id1><bcs1><bc>            in   <decl switchid> is
        <id1><bcs1>                in   <decl switchid> co

15.11   formal  <id1><bcs1> -->  <id1><bcs1>     in   <decl label> co
15.12   formal  <id1><bcs1> -->  <id1><bcs1>     in   <decl switchid> co

15.13   label  <label1><bcs1> -->  <label1><bcs1>  in   <decl label> co

15.14   switch  <id1><bcs1> -->  <id1><bcs1>  in   <decswitchid> co

15.15   <label>          in   <sdexp> co
15.16   <switchdes>      in   <sdexp> co
15.17   (<dexp>)         in   <sdexp> co

15.18   <sdexp>                                              in   <dexp> co
15.19   if  <bexp>  then  <sdexp>  else  <dexp>  in  <dexp>co

15.20   <decl label>                 in   <decl sdexp> co
15.21   <decl switchdes>             in   <decl sdexp> co
15.22   (<decl dexp>)                in   <decl sdexp> co

15.23   <decl sdexp>     in   <decl dexp> co

15.24   if  <decl bexp>  then  <decl sdexp>  else  <decl dexp>
        in  <decl dexp> co

15.25   <switchid>[<subexp>]  in  <switchdes> co

15.26   <decl switchid>[<decl subexp>]    in   <decl switchdes> co
```

16.1 <saexp> in <sexp> co

16.2 <sbexp> in <sexp> co

16.3 if <bexp1> then <sexp1> else <exp1> is if va (<bexp1>) then <sexp1> else <exp1> co

16.4 if true then <sexp1> else <exp> is <sexp1> co

16.5 if false then <sexp> else <exp1> is <exp1> co

16.6 (<aexp1>) is <aexp1> co

16.7 (<bexp1>) is <bexp1> co

16.8 <saexp1><relop1><saexp2> is va (<saexp1>)<relop1> va (<saexp2>) co

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

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

16.11 <bterm1> ∨ <bfactor1> is va (<bterm1>) ∨ va (<bfactor1>) co

16.12 <implication1> ¬ <bterm1> is va (<implication1>) ¬ va (<bterm1>) co

16.13 <sbexp1> = <implication1> is va (<sbexp1>) = va (<implication1>) co

16.14 ¬ false co

16.15 ¬ true is false co

16.16 true ∧ true co

16.17 true ∧ false is false co

16.18 false ∧ true is false co

16.19 false ∧ false is false co

16.20 true ∨ true co

16.21 true ∨ false co

16.22 false ∨ true co

16.23 false ∨ false is false co

16.24 true = true co

16.25 false = false co

16.26 true = false is false co

16.27 false = true is false co

16.28 true ¬ true co

16.29 true ¬ false is false co

16.30 false ¬ true co

16.31 false ¬ false co

```
16.32   <in1>  <  <in2>  is  ¬ <in2>  ≤  <in1> co
16.33   <in1>  ≥  <in2>  is    <in2>  ≤  <in1> co
16.34   <in1>  >  <in2>  is  ¬ <in1>  ≤  <in2> co

16.35   <in1>  =  <in2>  is    <in1>  ≤  <in2>  ∧  <in2>  ≤  <in1> co

16.36   <in1>  ≠  <in2>  is  ¬ <in1>  =  <in2> co

16.37   <in1>  ≤  <in2>  is  va  (  <in1> - <in2> )  ≤  0 co

16.38   -<ui>    ≤  0  co
16.39    <ui>    ≤  0  is  false co
16.40    <ze>    ≤  0  co

16.41   +-<ui1>  is  -<ui1> co
16.42   --<ui1>  is   <ui1> co

16.43   <in1> +- <ui1>  is  <in1> - <ui1> co
16.44   <in1> -- <ui1>  is  <in1> + <ui1> co
16.45   <in1> -+ <ui1>  is  <in1> - <ui1> co

16.46   <factor1> ↑ <primary1>  is <factor1> ↑ va ( <primary1> ) co

16.47   <term1><multop1><factor1>  is
        va ( <term1> ) <multop1> va (  < factor1>)  co

16.48   <pm1><term1>  is <pm1> va ( <term1> )  co

16.49   <saexp1><pm1><term1>  is
        va (  <saexp1> ) <pm1> va ( <term1> )  co

16.50   <factor1> ↑ <ui1> is va ( <factor1> ↑ va ( <ui1>-1 ) ) × <factor1> co

16.51   <factor1> ≠ 0  -->  <factor1> ↑ 0  is  1 co

16.52   <in1> :  - <ui1>  is  - va ( <in1> : <ui1> ) co
16.53   <ui1> :    <ui2>  is 1 + va (  va ( <ui1>-<ui2> ) :  <ui2> )  co

16.54   <ui1>  <  <ui2>  -->  <ui1>  :  <ui2>  is  0 co

16.55   <in1> × - <ui1>  is  - va ( <in1> × <ui1> )  co
16.56   <ui1> ×   <ui2>  is  va ( <ui1> × va ( <ui2> -1 )  )  +  <ui1> co
16.57   <ui>  ×      0    is  0 co

16.58   0  in  <di>  co
16.59   1  in  <di>  co
16.60   2  in  <di>  co
16.61   3  in  <di>  co
16.62   4  in  <di>  co
16.63   5  in  <di>  co
16.64   6  in  <di>  co
16.65   7  in  <di>  co
16.66   8  in  <di>  co
16.67   9  in  <di>  co
```

16.68 <di> in <ui> co

16.69 <ui><di> in <ui> co

16.70 <ui> in <in> co

16.71 <pm><ui> in <in> co

16.72 0 in <ze> co

16.73 <ze>0 in <ze> co

16.74 -<ui1> + <ui2> is <ui2> - <ui1> co

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

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

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

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

16.79 <ui1>0 + <di2> is <ui1><di2> co

16.80 <di1> + <ui2>0 is <ui2><di1> co

16.81 <ui1>0 - <di2> is va (<ui1> - 1) 0 + va (10 - <di2>) co

16.82 10 - <di2> is 9 - va (<di2> - 1) co

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

16.84 <ui1><pm><ze> is <ui1> co

16.85 <ze> + <ui1> is <ui1> co

16.86 <ze> - <ui1> is - <ui1> co

16.87 <ze><pm><ze> is 0 co

16.88 0 + 1 is 1 co

16.89 1 + 1 is 2 co

16.90 2 + 1 is 3 co

16.91 3 + 1 is 4 co

16.92 4 + 1 is 5 co

16.93 5 + 1 is 6 co

16.94 6 + 1 is 7 co

16.95 7 + 1 is 8 co

16.96 8 + 1 is 9 co

16.97 9 + 1 is 10 co

16.98 (<di1> + 1 is <di2>) --> (<di2> - 1 is <di1>) co

17.1	<let>	in	<id>	co
17.2	<id><let>	in	<id>	co
17.3	<id><di>	in	<id>	co
17.4	a	in	<let>	co
17.5	b	in	<let>	co
17.6	c	in	<let>	co
17.7	d	in	<let>	co
17.8	e	in	<let>	co
17.9	f	in	<let>	co
17.10	g	in	<let>	co
17.11	h	in	<let>	co
17.12	i	in	<let>	co
17.13	j	in	<let>	co
17.14	k	in	<let>	co
17.15	l	in	<let>	co
17.16	m	in	<let>	co
17.17	n	in	<let>	co
17.18	o	in	<let>	co
17.19	p	in	<let>	co
17.20	q	in	<let>	co
17.21	r	in	<let>	co
17.22	s	in	<let>	co
17.23	t	in	<let>	co
17.24	u	in	<let>	co
17.25	v	in	<let>	co
17.26	w	in	<let>	co
17.27	x	in	<let>	co
17.28	y	in	<let>	co
17.29	z	in	<let>	co

17.30	A	in <let> co
17.31	B	in <let> co
17.32	C	in <let> co
17.33	D	in <let> co
17.34	E	in <let> co
17.35	F	in <let> co
17.36	G	in <let> co
17.37	H	in <let> co
17.38	I	in <let> co
17.39	J	in <let> co
17.40	K	in <let> co
17.41	L	in <let> co
17.42	M	in <let> co
17.43	N	in <let> co
17.44	O	in <let> co
17.45	P	in <let> co
17.46	Q	in <let> co
17.47	R	in <let> co
17.48	S	in <let> co
17.49	T	in <let> co
17.50	U	in <let> co
17.51	V	in <let> co
17.52	W	in <let> co
17.53	X	in <let> co
17.54	Y	in <let> co
17.55	Z	in <let> co
17.56	sign	in <let> co
17.57	dummy	in <let> co
17.58	<bcs><fas>l	in <let> co
17.59	<bcs><fas>m	in <let> co
17.60	true	in <logicalvalue> co
17.61	false	in <logicalvalue> co
17.62	<logicalvalue1>	is ≮ <logicalvalue1> ≯ co
17.63	–<ui1>	is ≮ –<ui1> ≯ co
17.64	+<ui1>	is ≮ <ui1> ≯ co
17.65	<ui1>	is ≮ <ui1> ≯ co
17.66	<ze>	is ≮ 0 ≯ co
17.67	<pm><ze>	is ≮ 0 ≯ co
17.68	o --> <name>	is ≮ o ≯ co
17.69	o	is ≮ o ≯ co

CHAPTER 4.

On the next pages we define three examples of the proposals for ALGOL X.
It seems probable that most of the other suggestions can also be described without too much trouble. However, some of the more radical changes, such as environment enquiries and input/output conventions have not yet been thorougly studied.
The given examples are not self-contained, since in many places they have to be extended with truths from chapter 3; we have rendered only the essential features of the proposals.

A.B. refers to the ALGOL Bulletin.

Definition of some proposals for Algol X :

1. Bit patterns and operations on them , P. Naur , A.B. 19.3.11.3.

X1.1 0 in <zo> co
X1.2 1 in <zo> co

X1.3 b <zo> in <patternconstant> co
X1.4 <patternconstant><zo> in <patternconstant> co

X1.5 <patternconstant> in <patternprimary> co
X1.6 <patternvariable> in <patternprimary> co
X1.7 (<patternexp>) in <patternprimary> co
X1.8 <patternfuncdes> in <patternprimary> co

X1.9 <patternprimary> in <patternsecondary> co
X1.10 ¬ <patternprimary> in <patternsecondary> co

X1.11 <patternsecondary> in <patternfactor> co
X1.12 <patternfactor> ∧ <patternsecondary> in <patternfactor> co

X1.13 <patternfactor> in <simplepattern> co
X1.14 <simplepattern> ∨ <patternfactor> in <simplepattern> co
X1.15 pattern <saexp> in <simplepattern> co
X1.16 <patternprimary> shift <saexp> in <simplepattern> co

X1.17 <simplepattern> in <patternexp> co

X1.18 if <bexp> then <simplepattern> else <patternexp>
in <patternexp> co

X1.19 <saexp> in <aexp> co
X1.20 <patternprimary> extract <primary> in <aexp> co
X1.21 if <bexp> then <saexp> else <aexp> in <aexp> co

X1.22 ¬ <patternprimary1> is ¬ va (<patternprimary1>) co

X1.23 ¬ <patternconstant1> 0 is va (¬ <patternconstant1>) 1 co
X1.24 ¬ <patternconstant1> 1 is va (¬ <patternconstant1>) 0 co

X1.25 ¬ b 0 is b 1 co
X1.26 ¬ b 1 is b 0 co

X1.27 <patternfactor1> ∧ <patternsecondary1> is
va (<patternfactor1>) ∧ va (<patternsecondary1>) co

X1.28 <patternconstant1> 0 ∧ <patternconstant2> 0 is
va (<patternconstant1> ∧ <patternconstant2>) 0 co

X1.29 <patternconstant1> 0 ∧ <patternconstant2> 1 is
va (<patternconstant1> ∧ <patternconstant2>) 0 co

X1.30 <patternconstant1> 1 ∧ <patternconstant2> 0 is
va (<patternconstant1> ∧ <patternconstant2>) 0 co

X1.31 <patternconstant1> 1 ∧ <patternconstant2> 1 is
va (<patternconstant1> ∧ <patternconstant2>) 1 co

X1.32 b 0 ∧ <patternconstant1> 0 is <patternconstant1> 0 co
X1.33 b 0 ∧ <patternconstant1> 1 is <patternconstant1> 0 co

X1.34 <patternconstant1> 0 ∧ b 0 is <patternconstant1> 0 co
X1.35 <patternconstant1> 1 ∧ b 0 is <patternconstant1> 0 co

X1.36 b 1 ∧ <patternconstant1> is <patternconstant1> co

X1.37 <patternconstant1> ∧ b 1 is <patternconstant1> co

X1.38 b 1 ∧ b 1 is b 1 co
X1.39 b 1 ∧ b 0 is b 0 co
X1.40 b 0 ∧ b 1 is b 0 co
X1.41 b 0 ∧ b 0 is b 0 co

X1.42 <simplepattern1> ∨ <patternfactor1> is
va (<simplepattern1>) ∨ va (<patternfactor1>) co

X1.43 <patternconstant1> 0 ∨ <patternconstant2> 0 is
va (<patternconstant1> ∨ <patternconstant2>) 0 co

X1.44 <patternconstant1> 0 ∨ <patternconstant2> 1 is
va (<patternconstant1> ∨ <patternconstant2>) 1 co

X1.45 <patternconstant1> 1 ∨ <patternconstant2> 0 is
va (<patternconstant1> ∨ <patternconstant2>) 1 co

X1.46 <patternconstant1> 1 ∨ <patternconstant2> 1 is
va (<patternconstant1> ∨ <patternconstant2>) 1 co

X1.48 b 0 ∨ <patternconstant1> is <patternconstant1> co

X1.49 <patternconstant1> ∨ b 0 is <patternconstant1> co

X1.50 b 1 ∨ <patternconstant1> 0 is <patternconstant1> 1 co

X1.51 b 1 ∨ <patternconstant1> 1 is <patternconstant1> 1 co

X1.52 <patternconstant1> 0 ∨ b 1 is <patternconstant1> 1 co

X1.53 <patternconstant1> 1 ∨ b 1 is <patternconstant1> 1 co

X1.54 b 1 ∨ b 1 is b 1 co
X1.55 $\bar{b}$ 1 ∨ $\bar{b}$ 0 is $\bar{b}$ 1 co
X1.56 $\bar{b}$ 0 ∨ $\bar{b}$ 1 is $\bar{b}$ 1 co
X1.57 $\bar{b}$ 0 ∨ $\bar{b}$ 0 is $\bar{b}$ 0 co

X1.58 pattern <saexp1> is pattern va (<saexp1>) co

X1.59 pattern - <ui> is o co

X1.60 pattern <ui1> is va (pattern <ui1> : 2) 1 co

X1.61 <ui1> : 2 × 2 = <ui1> -->
pattern <ui1> is va (pattern <ui1> : 2) 0 co

X1.62 <ui1> = 1 --> pattern <ui1> is b 1 co
X1.63 <ui1> = 0 --> pattern <ui1> is $\bar{b}$ 0 co

X1.64 <patternprimary1> shift <saexp1> is
va (<patternprimary1>) shift va (<saexp1>) co

X1.65 b <zo> shift - <ui> is o co

X1.66 <patternconstant1><zo> shift - <ui1> is
<patternconstant1> shift - <ui1 > + 1 co

X1.67 <patternconstant1> shift <ui1> is
<patternconstant1> 0 shift <ui1> - 1 co

X1.68 <in1> = 0 --> <patternconstant1> shift <in1> is
<pattternconstant1> co

X1.71 <patternprimary1> extract <primary1> is
va (<patternprimary1>) extract va (<primary1>) co

X1.72 <patternprimary> extract - <ui> is o co

X1.73 <patternconstant1> 0 extract <ui1> is
2 × (<patternconstant1> extract (<ui1> - 1)) co

X1.74 <patternconstant1> 1 extract <ui1> is
2 × (<patternconstant1> extract (<ui1> - 1)) + 1 co

X1.75 <ui1> = 1 -->
<patternconstant1><zo1> extract <ui1> is <zo1> co

X1.76 b <zo> extract <in> is o co

X1.77 <ui1> = 1 --> b <zo1> extract <ui1> is <zo1> co

Remarks:

1. We introduced some changes in the syntax.
2. We do not require that the number of bits in a pattern be constant during one execution of a program.
3. In the definition of the shift operations we took some arbitrary decisions:
   3.1 a shift to the right is defined as a clear shift.
   3.2 in a shift to the left zeroes are added at the right hand side of the bit pattern.
4. We assume that one does not want to define the operator pattern for negative argument.

2. Case expressions . C.A.R. Hoare , A.B. 18.3.7.

```
X2.1   case      <aexpl>             of  ( <aexplist1> ) is
       case va (  <aexpl> )          of  ( <aexplist1> ) co

X2.2   case      <in>                of  ( <aexp> )  is  o  co

X2.3   case      <in1>               of  ( <aexp> else <aexplist1> ) is
       case      <in1> - 1           of  ( <aexplist1> ) co

X2.4   <in1> = 1   -->
       case      <in1>               of  ( <aexp1>)  is  <aexp1> co

X2.5   <in1>  =  1 -->
       case      <in1>               of  (<aexp1> else <aexplist>) is
       <aexp1> co
```

Remarks.

1. For the syntax we refer to A.B. 18.3.7.
2. Lists of boolean expressions and of statements can be treated similarly.

3. Non rectangular arrays. P.Naur , A.B. 18.3.9.7.

```
X3.1    [<idlist>]<aexp>  in  <subscriptpart> co

X3.2    <subscriptpart><subscriptpartlist>          in  <subscriptpartlist> co
X3.3    <subscriptpart>                             in  <subscriptpartlist> co

X3.4    <bcs1><dbcs> -->  <id1>[<subexplist1>] is
                          <id1><bcs1>[ va ( <subexplist1>) ] co

X3.5    <id>bc [<subexplist>]  is  o co

X3.6    <id1><bcs1><bc>[<subexplist1>] is
        <id1><bcs1>    [<subexplist1>] co

X3.7    <id1><bcs1>  op2  ( <id2> / <bcs2><dbcs> ) -->
        <id1><bcs1> [<subexplist1>] is <id2><bcs2>[<subexplist1>] co

X3.8    <type>array <id1><bcs1>[<bpair>]<subscriptpartlist> -->
        <id1><bcs1>[<subexplist1>]  is
        <id1><bcs1>[<subexplist1>]<subscriptpartlist1> co

X3.9    <type> array  <id1><bcs1>[<bpair>]<subscriptpartlist> -->
        <id1><bcs1>[<subexp>]  is  o co

X3.10   <id1><bcs1>[<subexplist1>]<subscriptpart><subscriptpartlist1> is
        <id1><bcs1>[<subexplist1>]                <subscriptpartlist1> co

X3.11   <id><bcs>[<subexplist>]<subscriptpart>   is  o co

X3.12   <subexplist1>  op20 <idlist1> -->
        <id1><bcs1>[<subexplist1>][<idlist1>]<aexp1><subscriptpartlist1> is
        ( <idlist1> op21 <subexplist1> co <id1><bcs1>[va ( <aexp1> ) ] co

X3.13   <subexp> , <subexplist1>  op20  <id> , <idlist1>  is
        <subexplist1>  op20 <idlist1> co

X3.14   <subexp>  op20  <id>  co

X3.15   <id1>  op21  <aexp1>  is  <id1> :=  <aexp1> co

X3.16   <id1>,<idlist1>  op21  <aexp1>,<aexplist1>  is
        ( <id1>  op21  <aexp1> co  <idlist1> op21 <aexplist1> )  co
```

Remarks.

1. These truths should be extended with a suitable selection from the truths defining Algol 60.
2. It is easy to extend this definition by combination with the treatment of Algol 60 arrays to declarations of non rectangular arrays and to assignment to a subscripted variable.
3. Probably this also holds for extension to <u>own</u> non rectangular arrays.
4. We would prefer not to forbid the occurrence of type lists with the same number of simple variables in one array segment. As in many similar cases in Algol 60 this only complicates the description whereas it is perfectly well-defined what happens if the programmer introduces two or more type lists in one array segment : the mechanism of the processor selects the first one.
5. For X3.15 to be useful it is necessary to assume - perhaps implicit- <u>integer</u> declarations for the entries in the identifier list of a subscript part.