TECHNICAL REPORT

TR 25.088
13 December 1968

FORMAL DEFINITION OF ALGOL 60

P. LAUER

IBM LABORATORY VIENNA

18 April 1969

ERRATA SHEET

for TR 25.088, "Formal Definition of Algol 60", by P.E.Lauer

Page Formula Line (⟵ = "to be replaced by")

5-2 2 : hyphon ⟵ hyphen

5-6 (11) : priorities should be as follows

1 ⟵ 7
2 ⟵ 8
3 ⟵ 9
4 ⟵ 1
⋮
9 ⟵ 6

5-9 (25) : BLOCK[t] ⟵ BLOCK[t] ⟹

5-9 (26) 1 : DECL[t] ⟵ DECL[t] ⟹ λ

2 : DECL[t] ⟹ λ ⟵ DECL[t] ⟹

5-12 (61) : to be replaced by the following schema:

is-1-dummy-st∘s-else-st(t) & ¬ is-1-cond-st∘s-then-st(t):
CONDST[t] ⟹
if EXPR[s-decision(t)] then LST[s-then-st(t)]
¬(is-1-cond-st ∨ is-1-for-st)∘s-then-st(t):CONDST[t] ⟹
if EXPR[s-decision(t)] then LST[s-then-st(t)]
else LST[s-else-st(t)]

5-13 (64) 13 : lsit ⟵ list

: (1 ≤ i ≤ length ⟵ ($\forall$ i)(1 ≤ i ≤ length

17 : EXPRL[t] ⟵ EXPR[t]

IBM LABORATORY VIENNA, Austria

FORMAL DEFINITION OF ALGOL 60

by

P.E. LAUER

ABSTRACT

This report constitutes a formal definition of the syntax and semantics of the algorithmic language ALGOL 60. The method is based on an abstract syntax specifying programs in an abstract form. These are interpreted by means of an abstract machine characterized by the set of the states it can assume and its state transition function. An abstract program specifies an initial state of the machine and the subsequent behaviour of the machine is said to define the interpretation of the given abstract program. The concrete representation of abstract programs as character strings is achieved by means of a representation system.

Locator Terms for IBM Subject Index

ALGOL 60
Formal Definition
Syntax
Semantics
21 PROGRAMMING

TR 25.088
13 December 1968

C O N T E N T S

Page

PREFACE

This report constitutes a complete formal definition of the programming language ALGOL 60 as specified by the "Revised Report on the Algorithmic Language ALGOL 60" /1/ and replaces the two Lab Reports /3/ and /4/. The method and notation used in this formal definition was developed by the IBM Laboratory Vienna, Austria, in the process of completing a formal definition of PL/I. Full historical details about the origins and development of the method can be found in the "Method and Notation for the Formal Definition of Programming Languages" TR 25.087 /2/, where the general aspects of the method, i.e., those not depending on its application to any particular programming language were developed in detail. A knowledge of /2/ is presupposed and the method used in the present report is only explained with respect to its departures, either by generalization or specialization, from the method presented there.

ACKNOWLEDGEMENT

Thanks are due to Mr. P. LUCAS for the many clarifying discussions especially with respect to the general form of the present report. Special thanks to Dr. K. ALBER, Dr. H. BEKIĆ and Mr. M. FLECK who have both contributed considerably to the present report in the form of discussions and concrete suggestions for modelling specific aspects of ALGOL 60. Thanks also to Mr. C. JONES for his careful reading of the entire manuscript and his helpful suggestions with respect to the final form of the report.

REFERENCES

/1/ NAUR, P. (Ed.): Revised Report on the Algorithmic Language ALGOL 60.-
Comm. ACM, Vol. 6 (1963) No. 1; pp. 1-23.

/2/ LUCAS, P., LAUER, P., STIGLEITNER, H.: Method and Notation for the Formal Definition of Programming Languages.-
IBM Laboratory Vienna, Techn. Report TR 25.087 (June 1968).

/3/ LAUER, P.: Abstract Syntax and Interpretation of ALGOL 60.-
IBM Laboratory Vienna, Lab. Report LR 25.6.001 (April 1968).

/4/ LAUER, P.: Concrete Representation of Abstract ALGOL 60 Programs.-
IBM Laboratory Vienna, Lab. Report LR 25.6.002 (May 1968).

1. INTRODUCTION

The basis of the formal definition of ALGOL 60 presented in this report is an abstract syntax. This consists of the set of predicate definitions given in chapter 2. The predicates characterize a class of abstract objects which is identified with a class of abstract programs.

There is a mapping from the class of abstract programs to the class of corresponding concrete representations as character strings producible by the syntactic rules in Backus Normal Form (BNF rules, for short) given in /1/. This mapping is realized by means of a representation system as defined in section 3.2.1 of /2/. The specific representation system given in chapter 5 of the present report does not yield concrete representations for all the abstract programs. This is so because the abstract syntax defined in chapter 2 is wider than the concrete syntax given in /1/ (e.g., the former permits expressions other than boolean in conditional statements and other than designational expressions in goto statements). The set of possible concrete representations defined by the representation system is precisely the set of programs producible by means of the syntax rules of /1/.

Abstract programs are interpreted by means of an abstract sequential machine of the type defined in chapter 4 of /2/. The particular abstract machine, the socalled ALGOL 60 machine, is defined in chapter 3 of the present report. Furthermore, the notion of a computation and the nature and function of the different state components of the machine are described intuitively. Chapter 4 constitutes the interpretation of abstract programs by means of a definition of the machine behaviour corresponding to the programs. However, not all abstract programs that have concrete representations have an interpretation. It is well known that there are strings producible by means of the BNF rules, given in /1/, which are not (syntactically) correct ALGOL 60 programs; i.e., there are certain context dependencies (e.g., the matching of used identifiers with their corresponding declarations, etc.) which cannot be expressed by means of the BNF rules. These context dependencies are expressed, in /1/, as constraints formulated in ordinary English throughout the text. These additional constraints are realized in the interpreter presented in chapter 4, by means of explicit tests.

1.1 Notation and Conventions

Section 1.3 of /2/ contains a summary of the notational conventions and symbols taken over from various other theories, e.g., set theory, logic and arithmetic, etc. In the present section, any additional notations of a general nature used in this report are summarized and they will be used throughout the report without further comment. Any notation used in this report, not explained in this section or in section 1.3 of /2/, will have been explained in the body of /2/ and its occurrence in the present report will be followed by an appropriate reference to /2/.

1.1.1 Selectors

(1) is-elem-sel =

Note: This is the predicate characterizing the range of values of the function elem (cf. 2-17 /2/).

1.1.2 Arithmetic functions

The following metavariable satisfies the predicate :

x is-real-val

(2) sign(x) =

$x > 0 \longrightarrow 1$
$x = 0 \longrightarrow 0$
$x < 0 \longrightarrow -1$

(3) abs(x) =

x * sign(x)

(4) entier(x) =

$(\iota i)(\text{is-intg-val}(i) \;\&\; i \leq x < i+1)$

2. THE SYNTAX OF ABSTRACT PROGRAMS

Two basic types of syntax, viz. abstract syntax and concrete syntax, may be considered to be constitutive of the syntactic definition of a programming language. An abstract syntax specifies the programs of the language as to the structures significant for their subsequent interpretation and not as to how they are to be expressed for the purpose of communication either to oneself or others. A concrete syntax specifies the programs of the language as a set of character strings. Once the syntax has been given, i.e., once it is possible to determine what categories of well-formed programs a language is to have, one can ask questions as to the possible meanings to be assigned to them.

In this chapter an abstract syntax is presented. On the one hand, programs satisfying this syntax are mapped onto a set of corresponding concrete representations (i.e., character strings satisfying the BNF rules of /1/) by means of the replacement system defined in chapter 5. On the other hand, abstract programs are interpreted by means of an abstract sequential machine, the ALGOL 60 machine, in chapter 4.

Abstract programs are objects satisfying the predicate is-program which is defined in terms of various auxiliary predicates corresponding to the various sub-phrases of ALGOL 60 programs. The definitional method used is explained in detail in chapter 2 of /2/. The predicate definitions specifying the abstract syntax have been chosen as far as possible, so as to correspond to the main types of sub-phrases of ALGOL 60. Furthermore, if the predicate definitions are applied iteratively, then they describe the composition of a program in terms of its elementary components.

The elementary components of a program are the following elementary objects:

(a) A finite class of constant elementary objects, usually denoted by names written in capital letters (e.g. FOR, LABEL, T) including the special elementary object <> (emptylist).

(b) The infinite class of ALGOL 60 identifiers, characterized by the predicate is-id.

(c) The infinite class of real numbers (containing the class of all integer values characterized by is-intg-val), characterized by is-real-val.

(d) A finite set of ALGOL 60 basic symbols satisfying the predicate is-char.

Given a string str of capital letters, as mentioned in (a), one can construct a predicate is-str which is only satisfied by the object denoted by str.

is-str(x) = (x = str)

2.1 The Structure of Programs

A concrete ALGOL 60 program is either a block or a compound statement. Abstract programs are, however, always blocks. The reason for this is that the fictitious block, whose declaration part contains only the declarations of the standard functions together with the declarations of program local labels (if any), is always explicitly written around the abstract analogue of the concrete ALGOL 60 program. The reason for explicitly writing the fictitions block is that the interpretation of programs containing standard functions will be exactly the same as that of programs containing functions defined in the program itself. Furthermore, as already mentioned, the declaration part of the outermost block contains the declarations of program local labels since in the abstract program all labels are collected together into the declaration part of the innermost block containing the labeled statement.

(2.1) is-program = is-block

(2.2) is-block = (<s-decl-pt:is-decl-pt>,
<s-st-list:is-st-list>)

2.2 Declarations

By means of the declaration part of a block a set of names is introduced each of which is associated with a declaration. The declaration is a description of the properties the name is to have in the block in which it is being declared. A declaration part is an object which is constructed in such a way that the application of a name contained in it yields its declaration. If it occurs in a declaration part, a name is either an identifier or a selector satisfying the predicate is-elem-sel. The predicate is-elem-sel characterizes the range of the function elem(i) which is a function mapping integers onto a set of elementary selectors. This device is introduced into the abstract syntax of the declaration part of a block to accommodate the integer label feature of concrete ALGOL 60.

(2.3) is-decl-pt =

($\{$<name:is-decl> || is-id(name) ∨ is-elem-sel(name)$\}$)

Note: There is no specified order of the declarations and no factoring of attributes implied by the object satisfying the predicate is-decl-pt.

(2.4) is-decl =

is-var-decl ∨ is-proc-decl ∨ is-label-decl ∨ is-switch-decl

(2.5) is-var-decl =

(<s-scope:is-OWN ∨ is-Ω>,
<s-da:is-da>)

(2.6) is-proc-decl = (<s-type:is-type ∨ is-Ω>,
<s-par-list:is-id-list>,
<s-spec-pt:is-spec-pt>,
<s-body:is-st ∨ is-code>)

Note: If the procedure declaration is to define the value of a function designator then the type component of the declaration contains the type associated with the declared identifier, otherwise, it is Ω .

(2.7) is-spec-pt =

($\{$<id:is-spec> || is-id(id)$\}$)

Note: Only formal parameters called by value are specified, hence the fact that an identifier is contained in the specification part indicates that it is by value. All identifiers contained in the parameter list but not in the specication part are treated as by name parameters. Notice that the specification part may be Ω .

(2.8) is-spec =

is-type ∨ is-type-array ∨ is-LABEL

(2.9) is-type-array =

(<s-array-type:is-type>)

(2.10) is-code =

Note: In /1/ the formulation of procedure bodies in non-ALGOL code is permitted, but the code language is not specified in the reference language used in /1/, likewise, such specifications are left open in this document.

(2.11) is-label-decl = is-index-list

(2.12) is-index =

is-intg-val ∨ is-bool-val ∨ is-FOR

Note: A label declaration is a list of indices pointing to the statement which is considered to be labeled by means of the label declared in the declaration part. Each one of these indices is either a natural number pointing to a statement in a statement list, or a truth-value pointing either to the then or else alternative of a conditional statement, or the elementary object FOR pointing to the statement to be iterated by means of a for statement.

(2.13) is-switch-decl = is-expr-list

2.3 Data Attributes

Data attributes describe the range of a variable.

(2.14) is-da = is-type ∨ is-array

(2.15) is-type = is-arithm ∨ is-BOOL

(2.16) is-arithm = is-INTG ∨ is-REAL

(2.17) is-array = (<s-lbd:is-expr>,
<s-ubd:is-expr>,
<s-elem:is-da>)

Note: A two-dimensional array of scalars is considered as a one-dimensional array of one-dimensional arrays of scalars.

2.4 Statements

(2.18) is-st = is-block ∨ is-comp-st ∨ is-cond-st ∨ is-for-st ∨ is-goto-st ∨
is-proc-st ∨ is-assign-st ∨ is-dummy-st

(2.19) is-comp-st = is-st-list

(2.20) is-cond-st = (<s-decision:is-expr>,
<s-then-st:is-st>,
<s-else-st:is-st>)

Note: It is assumed that a conditional statement always has two components, i.e., if the corresponding conditional statement in concrete ALGOL 60 had no else alternative, the abstract conditional statement has a dummy statement.

(2.21) is-for-st = (<s-contr-var:is-var>,
<s-for-list:is-for-elem-list>,
<s-st:is-st>)

(2.22) is-for-elem = is-expr ∨ is-while-elem ∨ is-step-until-elem

(2.23) is-while-elem = (<s-init-expr:is-expr>,
<s-while-expr:is-expr>)

(2.24) is-step-until-elem = (<s-init-expr:is-expr>,
<s-step-expr:is-expr>,
<s-until-expr:is-expr>)

(2.25) is-goto-st = (<s-label:is-expr>)

(2.26) is-proc-st = (<s-id:is-id>,
<s-arg-list:is-arg-list>)

(2.27) is-arg = is-expr ∨ is-string

(2.28) is-string = is-string-elem-list

(2.29) is-string-elem = is-basic-symbol ∨ is-string

(2.30) is-basic-symbol =

Note: /1/ allows the handling of arbitary sequences of basic symbols, i.e., strings, used as actual parameters of procedures. The alphabet of basic symbols may vary, however, the nature of these characters will not be further elaborated in the present paper.

(2.31) is-assign-st = (<s-lp:is-var-list>,
<s-rp:is-expr>)

(2.32) is-dummy-st = is-DUMMY

2.5 Expressions

(2.33) is-expr = is-cond-expr ∨ is-infix-expr ∨ is-prefix-expr ∨
is-funct-ref ∨ is-var ∨ is-const

(2.34) is-cond-expr = (<s-decision:is-expr>,
<s-then-expr:is-expr>,
<s-else-expr:is-expr>)

(2.35) is-infix-expr = (<s-opr:is-infix-opr>,
<s-op-1:is-expr>,
<s-op-2:is-expr>)

(2.36) is-infix-opr = is-bool-infix-opr ∨ is-relat-infix-opr ∨
is-arithm-infix-opr

(2.37) is-bool-infix-opr = is-AND ∨ is-OR ∨ is-IMPL ∨ is-EQUIV

(2.38) is-relat-infix-opr = is-GT ∨ is-GE ∨ is-EQ ∨ is-LE ∨ is-LT ∨ is-NE

(2.39) is-arithm-infix-opr = is-ADD ∨ is-SUBTR ∨ is-MULT ∨ is-DIV ∨
is-INTGDIV ∨ is-POWER

(2.40) is-prefix-expr = (<s-opr:is-prefix-opr>,
<s-op:is-expr>)

(2.41) is-prefix-opr = is-NOT ∨ is-PLUS ∨ is-MINUS

(2.42) is-funct-ref = (<s-id:is-id>,
<s-arg-list:is-arg-list>)

(2.43) is-var = is-id ∨ is-subscr-var

(2.44) is-id =

Note: This predicate characterizes the infinite class of ALGOL 60 identifiers.

(2.45) is-subscr-var = (<s-id:is-id>,
<s-subscr-list:is-expr-list>)

(2.46) is-const = is-bool-const ∨ is-intg-const ∨ is-real-const

(2.47) is-bool-const = (<s-type:is-BOOL>,
<s-value:is-bool-val>)

(2.48) is-bool-val = is-T ∨ is-F

(2.49) is-intg-const = (<s-type:is-INTG>,
<s-value:is-intg-val>)

(2.50) is-intg-val =

Note: This predicate characterizes the class of all (positive, zero and negative) integer values. They are elementary objects and belong to the class characterized by is-real-val.

(2.51) is-real-const = (<s-type:is-REAL>,
<s-value:is-real-val>)

(2.52) is-real-val =

Note: This predicate characterizes the class of (real) numbers.

3. STATE COMPONENTS AND COMPUTATION OF THE ALGOL 60 MACHINE

The ALGOL 60 machine is an abstract sequential machine as described in chapter 4 of the "Method and Notation for the Formal Definition of Programming Languages" /2/. It is described by the set of all possible states which the machine can assume. This set is defined by the language function Λ, which applied to a given state yields a set of successor states, and by the set of possible initial states of the machine. An initial state of the machine contains, within one of its components, the abstract text to be interpreted. Any state from the set of possible states satisfies the predicate is-state:

(3.1) is-state = (<s-dn:is-dn>,
<s-un:is-intg-val>,
<s-d:is-d>,
<s-e:is-e>,
<s-ci:is-ci>,
<s-c:is-c>)

Abbreviations and Terms

For reference purposes abbreviations for major components of a given state ξ are introduced and will be used throughout the report. The terms given name the major state parts according to their content and use.

Component	Abbreviation	Term
s-dn(ξ)	DN	Denotation Directory
s-un(ξ)	UN	Unique Name Counter
s-d(ξ)	D	Dump
s-e(ξ)	E	Environment
s-ci(ξ)	CI	Control Information
s-c(ξ)	C	Control

3.1 The Computation of the Machine

Iterated application of the language function to a given initial state containing a given abstract text produces a sequence of states which the machine is said to assume. This sequence of states is the computation of the given text on the machine (chapter 4 of /2/).

The control C of a state of the machine governs the transition from that state to its successor state. C contains instructions which on execution, change the state as specified by the definition of the individual instructions.

There exist several places in ALGOL 60 where the sequence of operations is relevant but not specified by the language. Any choice of sequence for sequential execution is valid though it may result in different computations for different choices. This is why Λ yields a set of successor states. An example is the evaluation of operands in expressions.

3.1.1 The control C

The control part C of a state ξ of the ALGOL 60 machine contains a set of instructions. The control part is an abstract object described by a control representation (chapter 4, /2/). The instructions of a control part can be considered as arranged in the form of a control tree where each instruction may have a set of successor instructions and the instructions at the terminal nodes of the tree are candidates for immediate execution. The execution of such an instruction modifies the state of the machine as specified for the individual instruction (which may include changes to C).

An object of type control satisfies the predicate

(3.2) is-c .

3.1.2 The language function of the machine

The machine describes the interpretation of an ALGOL 60 program t by defining the set of possible computations resulting from the program. A computation is a sequence of states of the machine:

$$\xi_0, \xi_1, \xi_2, \cdots$$

satisfying the following two conditions:

(1) ξ_0 is an initial state corresponding to t as given by the function initial-state (cf. 4-2)

$$\xi_0 = \text{initial-state}(t)$$

(2) Each state ξ_{i+1} is produced by the language function Λ from its predecessor:

$$\xi_{i+1} \in \Lambda(\xi_i) \quad .$$

A computation is successful if it is finite:

$$\xi_0, \xi_1, \ldots, \xi_n \quad \text{and} \quad \Lambda(\xi_n) = \{\} \quad .$$

The language function Λ specifying the set of possible successor states of a given state ξ is defined below. Let χ be a selector which, when applied to the control component of ξ, yields the instruction to be executed next:

(3.3) $$\Lambda(\xi) = \{ \Psi(\xi, \chi) \mid \chi \in tn \circ s\text{-}c(\xi) \} \quad .$$

Functions Ψ and tn are formally defined in /2/ and merely listed here.

(3.4) $\Psi(\xi, \chi)$

Note: This function specifies the successor state relative to the execution of the instruction $\chi \circ s\text{-}c(\xi)$.

(3.5) tn(c)

Note: This function yields the set of all selectors which select a terminal instruction when applied to a control c. A terminal instruction is one which is a possible candidate for immediate execution.

3.2 Flow of Control

This section describes the state components of the machine which govern the flow of control. These components are the dump D and the control information CI.

3.2.1 The dump D

The dump serves to maintain a history of block activations as long as they are still active. It is essentially organized as a pushdown stack in order to guarantee the dynamic nesting of the block structure of the language.

All the information of the state of the machine that is valid only for the time of a particular block activation is contained in the four block-local state components: the dump D itself, the environment E, the control information CI, and the control C (cf. 3.1.1). These components must be accessible during the entire duration of a block activation; in particular, they have to be preserved during the lifetime of a nested block activation which may maintain its own block-local state components, in order to be available after termination of the nested block activation. Part of the information contained in these state components may be inherited by nested block activations, but it may not be reinherited by outer ones.

These requirements are satisfied by the dump mechanism. Whenever a new block activation is established, the block-local state components of the old one become components of the dump; thereby the block-local state components of the immediately preceding block activation become components of the dump component of the dump, etc.

Upon the termination of a block activation the topmost block-local state components of the dump are reinstalled in the state.

(3.6) is-d = is-Ω ∨ (<s-d:is-d>,
<s-e:is-e>,
<s-ci:is-ci>,
<s-c:is-c>)

Ref.: is-c (3.2)

3.2.2 The control information CI

The control information determines the flow of control within a single block activation and consists of the following three immediate components and satisfies is-ci:

TX = s-text(CI)	the text part
I = s-index(CI)	the index part
CD = s-cd(CI)	the control dump

(3.7) is-ci = is-Ω ∨ (<s-text:is-p-st-list ∨ is-p-cond-st ∨ is-p-for-st>,
<s-index:is-index ∨ is-Ω>,
<s-cd:is-cd>)

Note: The predicates is-p-[pred] in the text components are defined at 4-4.

The control dump consists of four immediate components, namely, the same three immediate components as a control information (i.e. text part, index part and control dump) and additionally a control. It satisfies the predicate is-cd which is defined as follows:

(3.8) is-cd = (<s-text:is-p-st-list ∨ is-p-cond-st ∨ is-p-for-st ∨ is-Ω>,
<s-index:is-index ∨ is-Ω>,
<s-cd:is-cd ∨ is-Ω>,
<s-c:is-c>)

The text component is, either a statement list and the index component an integer value pointing to some element of the list, or the text component is a conditional statement and the index component is a truth value pointing to one of its alternative component statements, or the text component is a for statement and the index component is the elementary object FOR pointing to the component statement which is to be iterated by means of the for statement.

It is the interpretation of conditional statements and for statements (i.e., statements possibly occurring as elements of statement lists and possibly themselves containing statement lists, conditional statements and for statements) which require the construction of the complicated flow of control mechanism. In particular, it is the occurrence of nested for statements which forces the construction of a dump mechanism CD similar to the one described for the interpretation of nested blocks, because of the interpretation of goto statements which are forbidden to transfer the control into a for statement (cf. 4.5.4).

Lastly, it is important to note that the interpretation of an element of a statement list or a component statement of a conditional statement or for statement may cause the termination of the current block activation (e.g. because of a goto statement leading out of the current block).

3.3 Associating Identifiers with Meaning

For each identifier declared in the declaration part of the text of a block, a unique name is created upon activation of the block. This unique name is associated with the corresponding identifier in the environment component of the state. The unique name is in turn associated in the denotation directory with the declaration corresponding to the identifier. Any additional information determing the meaning of an identifier is also entered under the unique name in the denotation directory (e.g. the current environment is added to the declaration of a procedure, or the depth of the current dump to the declaration of a label). The meaning of any identifier occurring in the text may be found in the following sequence: the application of the identifier to the environment yields the unique name which, when applied to the denotation directory, yields the denotation of the identifier.

3.3.1 Creation of unique names

Various situations requiring the use of unique names arise during the interpretation of programs. Unique names are elementary objects entering as components into state parts, but they are at the same time simple selectors by means of which components are inserted into state parts. Hence, they function as links between locations in the state, where they occur as objects, and the information which has been inserted by means of them while serving as selectors. Thereby the same information can be linked to different locations.

Unique names belong to an infinite ordered subset of the set of elementary objects. The only function of the unique name counter UN, is to keep count of the unique names already used. This prevents multiple use (or rather generation) of the same unique name.

The counter UN, together with the instruction un-name implements the generation of the unique names. The instruction makes use of the integer constituting the current UN and selects a unique name out of the set of unique names.

The definitions of the predicates characterizing the unique name counter, unique names and of the instruction un-name follows:

(3.9) is-un = is-intg-val

(3.10) is-n =

Note: This predicate characterizes the enumerably infinite set of unique names $n_0, n_1, n_2, \ldots$ which is contained in the set of elementary objects.

(3.11) un-name =

PASS:n_{UN}

s-un:UN+1

Note: For explanation of instruction definitions see chapter 4 of /2/.

3.3.2 The environment E

The environment serves to associate each declared identifier with a unique name. The same identifier declared in different block activations receives different unique names and thereby different meanings (except for own declared variables which are associated with the same unique name, viz., that produced by the prepass (cf. 4.2)). The environment is a block-local state component (cf. 3.2.1) and, hence, the current environment contains only unique names leading to the meanings of identifiers valid in the current block activation.

Everything that has been said concerning the environment must, however, be extended to include the handling of integer labels. Hence, in addition to the pairing of identifiers and unique names in the environment there occurs a pairing of selectors, corresponding to integer labels, with unique names. These selectors are from the range of the function elem(i) where i is an integer value. Given an integer label j, the function elem is applied to it and the resulting selector is associated in the environment with a unique name. The range of values of elem are characterized by the predicate is-elem-sel.

There is one further use made of the environment and that is the association of the unique name of a procedure identifier defining a function reference with the unique name (function value name) by means of which the value of the function reference may be stored or retrieved from the denotation directory. Hence, the environment may contain the pairing of unique names with unique names (cf. 4-10(19)).

The general term name has been used to refer to identifiers, selectors corresponding to integer labels, and unique names, all of which may be used as selectors in the environment component.

The environment component satisfies the predicate is-e defined as follows:

(3.12) is-e = ({<name:is-n> || is-name(name)})

(3.13) is-name = is-id ∨ is-elem-sel ∨ is-n

Ref.: is-id 2-6(44), is-n 3-6(10), is-elem-sel 1-2(1)

Environments are passed as arguments to instructions which interpret text in environments which may differ from the environment of the block activation current at the point of interpretation of the text. Such cases occur when the text is to be interpreted at a time when a nested block is active and has installed its own updated environment in the state (e.g., in the case of arguments which are passed by name to the formal parameters of a procedure).

3.3.3 The denotation directory DN

If the declaration of an identifier has already been interpreted in some block activation, the denotation directory will contain all the information necessary to determine the meaning of the identifier, except the value of an identifier declared as a variable. This value appears in the denotation directory only after a value has been assigned to the variable in the course of the interpretation of the executable text of the block. The information constituting the meaning of an identifier is called its denotation.

Denotations of identifiers vary for the different types of declarations of identifiers. The definition of the predicate characterizing denotation directories will be followed by definitions of the various types of denotation and an explanation of their components.

(3.14) is-dn = ({<n:is-den> || is-n(n)})

(3.15) is-den = is-var-den ∨ is-proc-den ∨ is-label-den ∨
is-switch-den ∨ is-name-par-den

Denotations of variables (including arrays) consist of a data attribute component and a value component. The former is constructed at the time of the interpretation of the declaration and is an evaluated data attribute. The latter is either a type (in the case of a simple variable) or (in the case of an array) an object containing information concerning the upper and lower bounds of the array. The value component is constructed at the time of an assignment of a value to the variable. Previous to the first assignment the component is Ω. Values can be simple (boolean or real) or array values. Array values are objects constructed from selectors satisfying the predicate is-elem-sel and other values (simple or array).

(3.16) is-var-den = (<s-da:is-eda>,
<s-value:is-value ∨ is-Ω>)

(3.17) is-eda = is-type ∨ is-array-eda

(3.18) is-array-eda = (<s-lbd:is-intg-val>,
<s-ubd:is-intg-val>,
<s-elem:is-eda>)

(3.19) is-value = is-simple-val ∨ is-array-val

(3.20) is-simple-val = is-bool-val ∨ is-real-val

(3.21) is-array-val = ({<elem(i):is-value> || is-intg-val(i)})

Procedure denotations are constructed at the time of the interpretation of the declaration of the procedure identifier and consist of the declaration of the procedure and an added environment component containing the environment current at the time of the interpretation of the declaration. It is this environment which will be used in the interpretation of the body of the procedure when the procedure is called.

(3.22) is-proc-den = (<s-type:is-type ∨ is-Ω>,
<s-par-list:is-id-list>,
<s-spec-pt:is-spec-pt>,
<s-body:is-st ∨ is-code>,
<s-e:is-e>)

A label denotation is constructed at the time of the interpretation of its declaration and consists of a static location which is identical with the index list from the label declaration, and a dynamic location which is an integer value indicating the depth of the dump at the time of the block activation within which the declaration of the label is being interpreted.

(3.23) is-label-den = (<s-stat-loc:is-index-list>,
<s-dyn-loc:is-intg-val>)

A switch denotation is constructed at the time of the interpretation of the declaration of the switch identifier and consists of this declaration together with the environment current at the time of the interpretation of the declaration.

(3.24) is-switch-den = (<s-switch-list:is-expr-list>,
<s-e:is-e>)

The last kind of denotation, which, however, does not originate from a declaration, is that of parameters specified by name. It is constructed at the time of the interpretation of a procedure call or the invocation of a function designator. It consists of the text of the argument corresponding to the by name parameter and the environment current at the time of the call or the invocation.

(3.25) is-name-par-den = (<s-text:is-arg>,
<s-e:is-e>)

4. THE INTERPRETATION OF ABSTRACT PROGRAMS

The behaviour of the ALGOL 60 Machine can be described formally by means of instruction definitions. The notational devices for defining instructions have been presented in /2/. The description of the machine behaviour can be conceived as constituting a definition of the semantics of ALGOL 60 programs. This behaviour is a function of the structure of the machine state (chapter 3), the abstract syntax of text (chapter 2) and the instructions corresponding to significant parts of abstract program text.

The transition from a given abstract program to the initial state of the interpreting machine is accomplished by the function initial-state. The language function Λ is then defined by specifying which instructions are required by the significant program parts.

The formal definition of the semantics of ALGOL 60 presented in this chapter is patterned after the above outline. The definition proper is preceded by a statement concerning the treatment of program errors by the semantic model, followed by a discussion of the initial actions of the interpreting machine including the definition of the function initial-state and the socalled prepass.

The definition of the instructions which perform the interpretation proper is divided into three major sections. The first part contains the definition of instructions causing the activation and deactivation of blocks. This involves the introduction of names (identifiers) and the devices for specifying the meaning they are to have within a block activation (declarations). These instructions constitute the interpretation of the block and procedure statement. The second part involves the definition of instructions influencing the flow of control, i.e. specifying the sequence in which statements are to be executed. These instructions are constitutive of the meaning of the compound statement (statement list), conditional statement, for statement and goto statement. The last part defines the instructions causing the manipulation and modification of data and this constitutes the interpretation of the assignment statement and models expression evaluation.

4.1 The Treatment of Program Errors

The language function Λ defines the set of successor states of a given state of the interpreting machine (see 3.2.1). This function is a partial function since there are states for which no successor states are defined. A program which gives rise to a computation leading to a state for which Λ is undefined is called erroneous.

A successor state of a given state is a function of one of the instructions which are candidates for execution in that state. A successor state is undefined if the relevant instruction is undefined. Conditional expressions allow the definition of partial functions by incomplete case distinctions. However, in the present report, the instruction error and the function error are used in conditional expressions to indicate undefined situations. On the one hand, the occurrence of error or error indicates to the reader

that the respective undefined situation actually may arise, whereas, in the case of a non-occurrence of these in conditional expressions, it is guaranteed that only the defined situations may arise when syntactically correct programs are interpreted. On the other hand, it is possible that the undefined (erroneous) case may appear anywhere in a list of case distinctions, e.g.

$$f(x) = p_1 \longrightarrow e_1$$
$$p_2 \longrightarrow \text{error}$$
$$p_3 \longrightarrow e_2$$

An instruction is undefined in a given instance, if its execution in some way leads to the execution of the instruction error or the evaluation of the function error, in that instance.

There are two additional instances which render an instruction undefined:

a) the application of Ω as a function to any argument,
b) meta-expressions of the form $(\iota x)(e(x))$, if $e(x)$ denotes T for no x, or for more than one x.

It is important to distinguish undefined successor states in this sense from successor states defined by implementation defined functions. Range and domain, or constraints for these functions are specified in the definition as far as they are known.

4.2 The Initial Actions of the Interpreting Machine

The state ξ of the machine consists of the immediate components listed in chapter 3, p. 3-1.

The underlined abbreviations for state components are used in defining instructions. The structure and use of these components have been described in chapter 3.

The computation starts with the initial state ξ_o. The transition from the given program t_o, i.e. an abstract object satisfying the predicate is-program, to the initial state ξ_o, is accomplished by the function initial-state. Essentially, the action of the function consists of setting the unique name counter to 0 and setting up the sole instruction int-program in the control. All other immediate components of the state ξ_o are Ω. Formally this can be expressed by the definitions

$$\xi_o = \text{initial-state}\ (t_o)$$

and

$$\text{initial-state}(t) = \mu_o(\langle \text{s-un:0} \rangle, \langle \text{s-c:int-program}(t) \rangle)$$

for: is-program(t)

The instruction definitions which now follow constitute the definition of the language function Λ, i.e. they indicate the possible transitions from one state of the machine to the next. In other words, they yield the interpretation of the language ALGOL 60 by specifying the meaning of any arbitrary ALGOL 60 program.

Before the interpretation proper, however, the socalled prepass is performed. This is indicated formally in the definition of int-program:

(4.1) int-program(t) =

int-block(pt);
pt:prepass-text(t)

for: is-program(t)

The prepass adds to each declaration of an OWN declared variable a unique name as its unique name component. In this way it is ensured that subsequent entries into blocks use one and the same unique name.

It should be noted that after the prepass all those parts of the program which may contain declarations need no longer satisfy the predicates given in the abstract syntax of program. Instead, corresponding predicates for the modified text are defined. The names of these predicates are of the form is-p-[pred] instead of is-[pred] (e.g., is-p-block instead of is-block).

The following definitions utilize as Metavariables:

t	is-program	the program to be interpreted
un		an auxiliary object containing for each declaration $\chi(t)$ its unique name component as $\chi(un)$
pt	is-p-program	the text of the program to be interpreted, after the completed prepass

(4.2) prepass-text(t)

prep-text-1(t,un);
$\{\chi(un):$un-name $\mid$ is-OWN$\circ$s-scope$\circ\chi(t)\}$

(4.3) prep-text-1(t,un) =

PASS:μ(t;$\{\langle$s-n$\circ\chi:\chi(un)\rangle \mid$ is-n$\circ\chi(un)\}$)

The following is the definition schema for predicates is-p-[pred] which apply to objects constituting parts of the given text as modified by the prepass:

$$\text{is-p-}[\text{pred}](\text{pt}) =$$
$$(\exists t)(\text{is-}[\text{pred}](t) \;\&\; t=\delta(\text{pt};\{\text{s-n}\circ\chi \mid \text{is-OWN}\circ\text{s-scope}\circ\chi(\text{pt})\})) \;\&$$
$$(\forall\chi)(\text{is-OWN}\circ\text{s-scope}\circ\chi(\text{pt}) \supset \text{is-n}\circ\text{s-n}\circ\chi(\text{pt}))$$

In the table below the corresponding predicates is-[pred] of the abstract syntax of program are given on the right hand side of the table:

modified predicate	predicate of the abstract syntax
is-p-var-decl	is-var-decl
is-p-decl	is-decl
is-p-decl-pt	is-decl-pt
is-p-block	is-block
is-p-st-list	is-st-list
is-p-st	is-st
is-p-comp-st	is-comp-st
is-p-for-st	is-for-st
is-p-cond-st	is-cond-st
is-p-proc-decl	is-proc-decl

4.3 Metavariables and Abbreviations

The metavariables used throughout the whole definition and the predicates characterizing their range are listed below. Metavariables concerning only one of the subsections of the definition will be similarly listed at the heading of the appropriate section. If the range of a metavariable x is specified, then this specification holds also for all metavariables of the form x-i, where i is a decimal digit.

Ranges of metavariables are significant in those cases where they are bound by a logical quantifier (including the descriptor ι) or by the implicit set notation. In all other cases the indication of the range has the character of a comment.

arg	is-arg	argument
arg-list	is-arg-list	argument list
body	is-st v is-code	body of a procedure declaration
cd	is-cd	control dump
d	is-d	dump
den	is-den	denotation
eda	is-eda	evaluated data attribute
env	is-e	environment
expr	is-expr	expression
i	is-intg-val	integer value
id	is-id	identifier

cont'd

index	is-index	index(either an integer value, T, F, or FOR)
indl	is-index-list	index list
lbd	is-intg-val	integer value constituting the lower bound of an array attribute
lp	is-var	variable used as left part of an assignment statement
lp-list	is-var-list	left part list
n	is-n	unique name
op	is-op	operand
opr	is-infix-opr v is-prefix-opr	operator
ref	is-ref	reference
ref-list	is-ref-list	reference list
sl	is-expr-list	expression list of a subscripted variable
t		text
truth	is-boolean-value	boolean value (truth value)
type	is-type	type
ubd	is-intg-val	integer value constituting the upper bound of an array attribute

Abbreviations are used in the formulas to retain perspicuity concerning their structuring. Names used as abbreviations are distinguishable by their subscripts, and subscripted names are solely used as abbreviations. Abbreviations are defined under the heading "where:" immediately following the formula in which they are used.

4.4 Interpretation of Blocks and Procedure Statements

Blocks and procedure statements are treated together in this section because they both establish new block activations and, hence, introduce new identifiers and specify their meaning by means of declarations. Since, however, they differ in certain details, they will themselves be treated separately within this section. The main difference between blocks and procedure statements is the fact that one and the same procedure can be called at different locations in the program whereas a block can only be executed (interpreted) at the location at which it is written in the program. But this is a difference which is not significant for our present consideration of the establishment of block activations. Significant differences from this point of view are the fact that the notions of parameter passing and the return of a value (as in the case of a function designator) do not apply to blocks. One further difference is the fact that procedures are named entities in ALGOL 60 and blocks are not. Since procedures (and function designators) are named entities, their identifiers (names) may be used in (or as) arguments of other procedures or function designators. In such cases they may be called in block-activations the environments of which differ from the environment which was set up at the time of the interpretation of the procedure declaration. The body of the procedure must, however, be interpreted in the latter environment and so it is this environment which is established as the

current environment. The arguments of the procedure are, however, evaluated in the environment current at the time of the call of the procedure or the invocation of the function designator.

Metavariables

code	is-code	code procedure body
da	is-da	data attribute
decl	is-decl	declaration
dn	is-dn	denotation directory
dp	is-decl-pt	declaration part
name	is-name	an identifier, a selector from the range of elem (where the corresponding argument is an integer label), or a unique name
par-list	is-id-list	formal parameter list
spec	is-spec	specifier
spec-pt	is-spec-pt	specification part

4.4.1 Blocks

A block activation is effected by the instruction int-block. The arguments of the instruction are program texts as modified by the prepass.

After dumping the block-local state components and establishing new initial block-local components, the following actions are taken: updating the environment by associating the locally declared identifiers with unique names, updating the denotation directory by appropriate entries from the declarations of the locally declared identifiers, interpreting the statement list of the block and, finally, reinstalling the block-local state components of the dynamically preceding block activation. These actions are taken as a result of the execution of appropriate instructions installed in the new control.

Own declared identifiers are associated with the unique names in the unique name component of their corresponding declarations (which were inserted by the prepass). Array bounds of own declared variables are recalculated at each new block entry.

(4.4) int-block(t) =

```
   s-d :stack(ξ)
   s-ci:Ω
   s-c:unstack ;
         int-st-list(s-st-list(t));
           int-decl-pt(s-decl-pt(t));
             update-env(s-decl-pt(t))
```

for : is-p-block(t)
Ref.: int-st-list 4-13(26)

(4.5) stack(ξ) =

μ_o(<s-d :s-d(ξ)>,
<s-e :s-e(ξ)>,
<s-ci:s-ci(ξ)>,
<s-c :s-c(ξ)>)

(4.6) update-env(dp) =

null;
{upd-name(name,n);
n:un-name | is-name(name) & ¬ is-Ω∘name(dp) &
is-Ω∘s-scope∘name(dp)} ∪
{upd-name(name,s-n∘name(dp)) | is-name(name) &
is-OWN∘s-scope∘name(dp)}

Ref.: un-name 3-6(11)

(4.7) upd-name(name,n) =

s-e:μ(E;<name:n>)

(4.8) null =

PASS:Ω

Note: Apart from being used for the interpretation of the dummy statement, this instruction is used to construct a control tree for instructions to be executed in an unspecified order (cf. /2/, p. 4-27).

(4.9) int-decl-pt(dp) =

null;
{int-decl(name,dp) | is-name(name) & ¬is-Ω∘name(dp)}

(4.10) int-decl(name,dp) =

is-p-var-decl∘name(dp) ⟶

upd-var-dn(name(E),eda);
eda:eval-da(s-da∘name(dp),δ(E;{name-1 | is-name(name-1) & name-1(dp)≠Ω}))

T ⟶

upd-dn(name(E),den);
den:mk-den(name(dp))

Note: If the declaration of an identifier is a variable declaration with array bounds which have to be evaluated, the environment utilized to do this is the current environment minus the identifiers which were declared in the declaration part presently being interpreted (see the statement in /1/, 5.2.4.2).

(4.11) eval-da(da,env) =

```
is-type(da) ——→ PASS: da
T ——→
   pass(eda);
     s-lbd(eda) :eval-intg-expr(s-lbd(da),env),
     s-ubd(eda) :eval-intg-expr(s-ubd(da),env),
     s-elem(eda):eval-da(s-elem(da),env)
```

(4.12) pass(x) =

```
PASS:x
```

(4.13) upd-var-dn(n,eda) =

```
s-dn:μ(DN;<s-da∘n:eda>)
```

(4.14) mk-den(decl) =

```
is-p-proc-decl(decl) ——→ PASS:μ(decl;<s-e:E>)
is-label-decl(decl)  ——→ PASS:μ0(<s-dyn-loc:depth(D)>,
                                 <s-stat-loc:decl>)
is-switch-decl(decl) ——→ PASS:μ0(<s-switch-list:decl>,
                                 <s-e:E>)
```

for : (is-p-proc-decl ∨ is-label-decl ∨ is-switch-decl)(decl)

Ref.: depth 4-20(49)

(4.15) upd-dn(n,den) =

```
s-dn:μ(DN;<n:den>)
```

for: (is-proc-den ∨ is-label-den ∨ is-switch-den)(den)

(4.16) unstack =

```
s-d :s-d(D)
s-e :s-e(D)
s-ci:s-ci(D)
s-c :s-c(D)
```

4.4.2 Procedure statements

The procedure statement is interpreted by the instruction int-proc-st. This instruction has three arguments, the first of which is the text of the procedure statement, the second is the environment of the block-activation in which the procedure was called and by means of which the actual parameters (arguments) are to be evaluated, and the third is the function value name which is the unique name associated with the procedure identifier

of a function designator used to access the resulting value of the function designator. In the case of a non-type procedure or a type procedure called by a procedure statement, the third argument is Ω. Before the interpretation proper, the function value name is installed in the denotation directory DN and the arguments of the called procedure or invoked function designator are installed. In many ways, the effect of the interpretation of a procedure statement on the state of the machine is similar to that of the interpretation of a block. The similarities are that the old local state components are stacked, that the control information is set to Ω and that the last instruction in the control is unstack which causes the termination of the block activation.

(4.17) int-proc-st(t,env,n) =

```
is-proc-den(final-den_t) ⟶
   s-d :stack(ξ)
   s-e :s-e(final-den_t)
   s-ci:Ω
   s-c :unstack;
         int-st(s-body(final-den_t));
           install-arg-list(s-par-list(final-den_t),
                            s-spec-pt(final-den_t),s-arg-list(t),env);
             install-funct(final-n_t,n)
T ⟶ error
```

where: $\text{final-den}_t = \text{final-n}_t(DN)$

$\text{final-n}_t = \text{final-n}(\text{s-id}(t)(env),DN)$

for : is-p-proc-st(t)

Ref. : int-st 4-15(32), unstack 4-8(16), stack 4-7(5)

Note : Error: if t is a by name parameter of some procedure p, and t is called by a procedure statement or invoked by a function designator in the body of p but the final denotation of t turns out to be other than a procedure denotation.

(4.18) final-n(n,dn) =

```
¬is-name-par-den∘n(dn) ⟶ n
is-id∘s-text∘n(dn) ⟶
   final-n(id_1(env_1),dn)
T ⟶ error
```

where: $id_1 = \text{s-text}\circ n(dn)$

$env_1 = \text{s-e}\circ n(dn)$

Note : Error: if n(dn) is a formal parameter denotation the text component of which is other than an identifier.

(4.19) install-funct(proc-n,funct-n) =

is-Ω(funct-n) & is-type•s-type•proc-n(DN) ⟶

upd-var-dn(n,s-type•proc-n(DN));
upd-name(proc-n,n);
n:un-name

is-Ω(funct-n) & is-Ω•s-type•proc-n(DN) ⟶ null

is-Ω•s-type•proc-n(DN) ⟶ error

T ⟶

upd-var-dn(funct-n,s-type•proc-n(DN));
upd-name(proc-n,funct-n)

for : is-n(proc-n) & (is-n ∨ is-Ω)(funct-n)

Ref.: upd-var-dn 4-8(13), upd-name 4-7(7), un-name 3-6(11)
null 4-7(8)

Note: This instruction installs the function value name of type procedures, invoked as function designators, in the environment and enters the appropriate data attribute in the denotation directory, utilizing the function value name and the type from the procedure declaration. In the case of a type procedure called as a procedure, a new unique name is generated, installed in the environment and associated with the type of the procedure in DN. This unique name is used in any assignment statement of the body having the procedure identifier as a left hand side. Since this unique name is not known outside the block activation caused by the procedure call the value assigned by means of it will be lost on exit from the procedure. For non-type procedures the instruction is equivalent to null.
Error: if a non-type procedure is invoked as a function designator.

(4.20) install-arg-list(par-list,spec-pt,arg-list,env) =

length(arg-list) = length(par-list) &
(∀i,j)(i≠j & 1 ≤ i ≤ length(par-list) ⊃ elem(i,par-list) ≠ elem(j,par-list)) &
(∀id)(is-id(id) & ¬ is-Ω•id(spec-pt) ⊃ (∃i)(id=elem(i,par-list))) ⟶

null;
{install-arg(n,par_i(spec-pt),arg_i,env);
upd-name(par_i,n);
n:un-name | 1 ≤ i ≤ length(par-list)}

T ⟶ error

where: par_i = elem(i,par-list)

arg_i = elem(i,arg-list)

Ref. : null 4-7(8), upd-name 4-7(7), un-name 3-6(11) , length 2-20/2/

Note : Error: if the number of argument positions in the parameter list of the procedure declaration is not the same as the number of argument positions

in the procedure statement or function designator; or if the formal parameter list contains two different occurrences of the same identifier; or if parameters not occurring in the parameter list are specified.

(4.21) install-arg(n,spec,arg,env) =

is-Ω(spec) ⟶

upd-dn(n,μ_o(<s-text:arg>,<s-e:env>))

is-type(spec) ⟶

convert-assign(ref_n,op);
op:eval-expr(arg,env),
upd-var-dn(n,spec)

is-type-array(spec) & is-id(arg) & is-array-eda∘s-da(final-den_{arg}) ⟶

assign-array(n,final-n_{arg},s-value,s-da(final-den_{arg}));
upd-var-dn(n,comb-da(s-da(final-den_{arg}),s-array-type(spec)))

is-LABEL(spec) ⟶

upd-dn(n,den);
den:eval-des-expr(arg,env)

T ⟶ error

where: $ref_n = \mu_o$(<s-type:spec>,
<s-value-sel:s-value∘n>)

final-n_{arg} = final-n(arg(env),DN)

final-den_{arg} = final-n_{arg}(DN)

Ref. : upd-dn 4-8(15), convert-assign 4-24(63), eval-expr 4-25(66), upd-var-dn 4-8(13), eval-des-expr 4-19(45)

Note : This function enters the appropriate denotation in the denotation directory for each argument occurring in the argument list of the procedure statement or function designator. If there is no corresponding specifier for an argument, this indicates that the argument is to be passed by name (i.e., all specifications of by name parameters are treated as comment in the present paper).
Error: if named entities are passed by value other than variables, labels or parameterless type procedures. In the last case the specification is the type of the procedure, and the case is handled like the by value case for simple variables (alternative two in the above instruction definition).

(4.22) comb-da(eda,type) =

is-type(eda) ⟶ type

T ⟶ μ(eda;<s-elem:comb-da(s-elem(eda),type)>)

(4.23) assign-array(par-n,arg-n,sel,eda) =

is-type(eda) ⟶

convert-assign(ref_{par},op);
op:get-op(ref_{arg})

T ⟶

null;
{assign-array(par-n,arg-n,value-sel(i,sel),s-elem(eda)) |
s-lbd(eda) ≤i ≤s-ubd(eda)}

where: ref_{par} = μ_o(<s-type:get-type∘s-da∘par-n(DN)>,
<s-value-sel:sel∘par-n>)

ref_{arg} = μ_o(<s-type:eda>,
<s-value-sel:sel∘arg-n>)

for : is-n(par-n) & is-n(arg-n)

Ref. : convert-assign 4-24(63), get-op 4-26(70), null 4-7(8)

(4.24) get-type(eda) =

is-type(eda) ⟶ eda
T ⟶ get-type∘s-elem(eda)

for: (is-da ∨ is-eda)(eda)

(4.25) int-code(code) =

Note: /1/ allows the formulation of procedure bodies in non-ALGOL 60 languages and leaves the use of this feature entirely up to the hardware representation. Hence, the interpretation of such code cannot be specified further in this report.

4.5 Interpretation of Statements Influencing the Flow of Control

The interpretation of ALGOL 60 statements influencing the flow of control is accomplished by defining instructions specifying the sequencing of interpretation of statements. On the one hand, the instruction corresponding to the statement list (regardless as to whether the list originated from a block or a compound statement) specifies a sequential order of interpretation of the component statements of the list, viz., the order in which they are written. On the other hand, the instruction corresponding to the conditional statement specifies which of its two statements is to be interpreted and the instruction corresponding to the for statement specifies repeated interpretation of its statements; both do so depending on certain specifiable conditions. Lastly, the instruction corresponding to the goto statement permits the transfer of the control to previously specified points in the program. Such transferral may cause the deactivation of blocks but not the activation of blocks.

The general device for modelling the transferral of the control in the various ways sketched above is the control information component of the machine state (cf. section 3.2.2).

It is only the inclusion of the goto statement in ALGOL 60 which requires that the interpretation of the conditional statement and the for statement involve the use of the control information. However, once the goto statement is included it seems more convenient to treat all instructions, involving the transfer of the control, by means of the control information device.

Metavariables:

for-elem	is-for-elem	for list element
for-list	is-for-elem-list	for list element list

4.5.1 Statement lists and single statements

As stated in section 3.2.2 the interpretation of statement lists is a function of the control information CI and the control part C. The text part s-text(CI) contains the entire text of the statement list elligible for interpretation. The index part s-index(CI) is an integer value indicating which statement of the list is currently being interpreted. The control part C contains solely the instruction which interprets this single statement, updates the index part and places the next statement in the control part for interpretation.

But even within one block activation there may be nested statement lists, component conditional statements and for statements (both of which may again contain statement lists, conditional statements, etc.). Hence, an additional component of the control information is needed to model this situation. This component is the control dump s-sc(CI) and it fulfills a similar function for nested statement lists, conditional statements and for statements, as the dump D fulfills for nested block activations. Text part, index part and control part indicate how the innermost nested statement list is to be interpreted whereas the control dump keeps track of the containing nested statement lists, conditional statements and for statements.

(4.26) int-st-list(t) =

is-<>(t) ⟶ null

T ⟶ upd-ci(1,t)

for : is-p-st-list(t)

Ref.: null 4-7(8)

(4.27) upd-ci(index,t) =

```
s-ci:μ₀(<s-text:t>,
        <s-index:index>,
        <s-cd:μ(CI;<s-c:C>)>)
s-c :int-st-1
```

for: is-p-st-list(t)

(4.28) int-st-1 =

```
int-next-st;
  int-st(take-st(s-index(CI),s-text(CI)))
```

(4.29) take-st(index,t) =

```
is-p-st-list(t) & is-intg-val(index) & 1≤index≤length(t) ⟶
  elem(index,t)
is-p-cond-st(t) & is-T(index)  ⟶ s-then-st(t)
is-p-cond-st(t) & is-F(index)  ⟶ s-else-st(t)
is-p-for-st(t) & is-FOR(index) ⟶ s-st(t)
T ⟶ error
```

for : (is-p-st-list ∨ is-p-cond-st ∨ is-p-for-st)(t)
Ref.: length 2-20/2/
Note: This function is a generalization of the function elem(i,t); it yields the statement out of t to which i points (cf. 2-20/2/).
Error: if used in the interpretation of a goto statement and a transfer to a non-existing statement or a transfer into a block is attempted.

(4.30) int-next-st =

```
is-intg-val∘s-index(CI) & s-index(CI)<length∘s-text(CI) ⟶
  int-st-1;
    upd-index
T ⟶
  s-ci:δ(s-cd(CI);s-c)
  s-c :s-c∘s-sd(CI)
```

Ref.: length 2-20/2/

(4.31) upd-index =

```
s-ci:μ(CI;<s-index:s-index(CI) + 1>)
```

(4.32) int-st(t) =

```
is-p-block(t)     ——→ int-block(t)
is-p-proc-st(t)   ——→ int-proc-st(t,E,Ω)
is-p-comp-st(t)   ——→ int-st-list(t)
is-p-cond-st(t)   ——→ int-cond-st(t)
is-p-for-st(t)    ——→ int-for-st(t)
is-goto-st(t)     ——→ int-goto-st(t)
is-assign-st(t)   ——→ int-assign-st(t)
is-dummy-st(t)    ——→ null
is-code(t)        ——→ int-code(t)
```

for : is-p-st(t)
Ref.: int-block 4-6(4), int-proc-st 4-9(17), int-assign-st 4-23(54), null 4-7(8), int-code 4-12(25)

4.5.2 Conditional statements

The conditional statement selects, depending on the value of an expression, one out of two statements for execution.

(4.33) int-cond-st(t) =

```
upd-ci(truth,t);
  truth:eval-truth(s-decision(t),E)
```

for : is-p-cond-st(t)
Ref.: upd-ci 4-14(27)

(4.34) eval-truth(expr,env) =

```
test-truth(op);
  op:eval-expr(expr,env)
```

Ref.: eval-expr 4-25(66)

(4.35) test-truth(op) =

```
is-BOOL∘s-type(op) ——→ PASS:s-value(op)
T ——→ error
```

Note: This instruction is the semantic equivalent of the ALGOL 60 syntactic restraint that only boolean expressions may occur at some specific point in the program (e.g. here in a conditional statement).

4.5.3 For statements

The for statement specifies iterated execution of a statement and the number of iterations is dependent on the values of various types of expressions.

```
(4.36) int-for-st(t) =

          iterate-for-list(ref,s-for-list(t),t);
            ref:eval-lp(s-contr-var(t),E)

       for : is-p-for-st(t)
       Ref.: eval-lp  4-23(57)

(4.37) iterate-for-list(ref,for-list,t) =

          is-<>(for-list) ⟶

            s-dn:δ(DN;s-value-sel(ref))

          T ⟶

            iterate-for-list(ref,tail(for-list),t);
              iterate-for(ref,head(for-list),t)

       for : is-p-for-st(t)
       Ref.: tail 2-20/2/,   head 2-20/2/
       Note: If the for list element list is exhausted, i.e., the iterative process,
             the for statement is left naturally (not by a goto statement), then the
             value of the control variable is undefined.

(4.38) iterate-for(ref,for-elem,t) =

          is-expr(for-elem) ⟶

            upd-ci(FOR,t);
              convert-assign(ref,op);
                op:eval-expr(for-elem,E)

          is-while-elem(for-elem) ⟶

            test-while(truth,ref,for-elem,t);
              truth:eval-truth(s-while-expr(for-elem),E);
                      convert-assign(ref,op);
                        op:eval-expr(s-init-expr(for-elem),E)

          is-step-until-elem(for-elem) ⟶

            iterate-for-1(ref,op,s-step-expr(for-elem),s-until-expr(for-elem),t);
              convert-assign(ref,op);
                op:eval-expr(s-init-expr(for-elem),E)

       for : is-p-for-st(t)
       Ref.: upd-ci  4-14(27),   convert-assign  4-24(63),   eval-expr 4-25(66),
             eval-truth  4-15(34)
```

(4.39) test-while(truth,ref,while-elem,t) =

truth ⟶ iterate-for(ref,while-elem,t);
upd-ci(FOR,t)

T ⟶ null

for : is-p-for-st(t) & is-while-elem(while-elem)
Ref.: upd-ci 4-14(27),null4-7(8)

(4.40) iterate-for-1(ref,op,step-expr,until-expr,t) =

test-until(truth,ref,step-expr,until-expr,t);
truth:eval-until(op,step-op,until-op);
step-op :eval-expr(step-expr,E),
until-op:eval-expr(until-expr,E)

for : is-p-for-st(t) & is-expr(step-expr) & is-expr(until-expr)
Ref.: eval-expr 4-25(66)

(4.41) eval-until(op,step-op,until-op) =

is-BOOL∘s-type(op) ∨ is-BOOL∘s-type(step-op) ∨
is-BOOL∘s-type(until-op) ⟶ error

$v_{step} > 0$ ⟶ PASS: infix-op(v_{op},v_{until},GT)

$v_{step} = 0$ ⟶ PASS:F

$v_{step} < 0$ ⟶ PASS: infix-op(v_{op},v_{until},LT)

where: v_{op} = s-value(op)
v_{step} = s-value(step-op)
v_{until} = s-value(until-op)

for : is-op(step-op) & is-op(until-op)
Ref.: infix-op 4-27(75)
Note: Error: if one of the arguments has a value of type Boolean and, hence, the arithmetic relational operations would be undefined.

(4.42) test-until(truth,ref,step-expr,until-expr,t) =

truth ⟶ null

T ⟶

iterate-for-1(ref,op,step-expr,until-expr,t);
convert-assign(ref,op);
op:eval-step(ref,step-expr);
upd-ci(FOR,t)

for : is-p-for-st(t) & is-expr(step-expr) & is-expr(until-expr)
Ref.: convert-assign 4-24(63), upd-ci 4-14(27)

```
(4.43) eval-step(ref,expr) =
          infix-op(op-1,op-2,ADD);
            op-1:get-op(ref),
            op-2:eval-expr (expr,E)
```

Ref.: infix-op 4-27(75), get-op 4-26(70), eval-expr 4-25(66)

4.5.4 Goto statements

The goto statement interrupts the sequential flow of control and transfers it to a location denoted by a label, where the sequential flow of control starts anew. For this purpose, the instruction corresponding to the goto statement must change the state of the machine in such a way that the resulting state is the same as the state which would have existed, had the labeled statement been reached from the beginning of the program by uninterrupted flow of control.

The localization of the statement denoted by a label, relative to its innermost containing block, is done statically by means of an index list consisting of integer values (pointing to elements of statement lists), truth values (pointing to alternatives of conditional statements) and the elementary object FOR (pointing to the component statement of for statements). This index list appears as the text of a label declaration in an abstract program.

The denotation of a label consists of two components, the statement location, which is the above mentioned index list and which localizes the labeled statement within one block activation; further, it consists of an integer value indicating the depth of the dump at the moment of the interpretation of the label declaration, i.e., uniquely characterizing the block activation within which the labeled statement pointed out by the label under consideration is to be found.

The interpretation of a goto statement is performed in the following four steps:

1) Close blocks until the current block activation is the one denoted by the dynamic location (dump depth) of the label denotation.

2) Reduce the control dump of the current block activation until the text component of the current control information contains, possibly nested, the labeled statement.

3) Build up appropriate levels in the control dump until the labeled statement is one of the statements of the current text part.

4) Adjust the index part to point to the labeled statement and continue sequential interpretation.

(4.44) int-goto-st(t) =

goto-l(den);
den:eval-des-expr(s-label(t),E)

for: is-goto-st(t)

(4.45) eval-des-expr(t,env) =

is-cond-expr(t) ⟶

eval-cond-des-expr(truth,s-then-expr(t),s-else-expr(t),env);
truth:eval-truth(s-decision(t),env)

is-subscr-var(t) & length∘s-subscr-list(t) = 1 & is-switch-den(final-den$_t$) ⟶

eval-switch(i,final-den$_t$);
i:eval-intg-expr(head∘s-subscr-list(t),env)

is-id(t) & is-name-par-den(den$_t$) ⟶

eval-des-expr(s-text(den$_t$),s-e(den$_t$))

is-id(t) & is-label-den(den$_t$) ⟶

PASS:den$_t$

is-intg-const(t) & is-label-den(elem∘s-value(t)(env)(DN)) ⟶

PASS:elem∘s-value(t)(env)(DN)

T ⟶ error

where: den$_t$ = t(env)(DN)

final-den$_t$ = final-n(s-id(t),DN)(DN)

for : is-expr(t) ∨ is-string(t) ∨ is-Ω(t)

Ref. : eval-truth 4-15(34), eval-intg-expr 4-24(59), length 2-20/2/

Note : This instruction differs from the instruction corresponding to non-designational expressions in that the length of the subscript list of a subscripted variable (i.e., a switch designator in concrete ALGOL 60) must be equal to 1, and in that the denotation of an integer constant declared as a label is not the constant itself but the label denotation associated with it.
Error: whenever the goto was to an undefined switch designator, since in that case t = Ω ; and also whenever the instruction was used in the installation of arguments during the interpretation of a procedure statement and the argument was a string.

(4.46) eval-cond-des-expr(truth,then-expr,else-expr,env) =

truth ⟶ eval-des-expr(then-expr,env)

T ⟶ eval-des-expr(else-expr,env)

for: is-expr(then-expr) & is-expr(else-expr)

(4.47) eval-switch(i,den) =

eval-des-expr(elem(i,s-switch-list(den)),s-e(den))

for : is-switch-den(den)

Note: This instruction eventually leads to an error when i is greater than the length of the switch list (see Note to (4.45)). The ALGOL report /1/ states that a goto to an undefined switch designator is equivalent to a dummy statement. The present alternative (leading to an error) was chosen since it would not be clear what would be meant by "equivalent to a dummy statement" in the case where a designational expression which was passed as a parameter to a procedure turned out to be undefined. If one wanted to model the equivalence to a dummy statement one would just add the state before the execution of the goto statement as an extra argument to the instruction eval-switch.

(4.48) goto-1(den) =

depth(D) = s-dyn-loc(den) ⟶ goto-2(s-stat-loc(den))

T ⟶

s-d :s-d(D)
s-e :s-e(D)
s-ci:s-ci(D)
s-c :goto-1(den)

for: is-label-den(den)

(4.49) depth(d) =

is-Ω(d) ⟶ 0

T ⟶ depth∘s-d(d) + 1

(4.50) goto-2(ind1) =

(∃ list)(list ≠ ⟨⟩ & ind1 = $list_{cd}$ ⌢ list) ⟶

goto-3((ιlist)(ind1 = $list_{cd}$ ⌢ list))

T ⟶

s-ci:δ(s-cd(CI);s-c)
s-c :goto-2(ind1)

where: $list_{cd}$ = index-list∘s-cd(CI)

(4.51) index-list(cd) =

is-Ω∘s-ci(cd) ⟶ ⟨⟩

T ⟶ index-list∘s-cd∘s-ci(cd) ⌢ ⟨s-index∘s-ci(cd)⟩

(4.52) goto-3(indl) =

```
is-FOR∘head(indl) & ¬is-FOR∘s-index(CI) ——→ error
length(indl) = 1 ——→ goto-4(head(indl))
T ——→
  s-ci:μ0(<s-text:take-st(head(indl),s-text(CI))>,
          <s-cd:μ(CI;<s-index:head(indl)>,
                    <s-c:int-next-st>)>)
  s-c :goto-3(tail(indl))
```

Ref.: int-st-1 4-14(28), int-next-st 4-14(30), head 2-20/2/, tail 2-20/2/, take-st 4-14(29)

Note: This instruction builds up the control information until the text part is the statement list, conditional statement or for statement, containing immediately the target statement of the goto. The first alternative excludes goto into a for statement. The second alternative is the basic case eventually to be arrived at: goto within the same level, just resetting s-index(CI) and continuing normally, i.e. with int-st-1. The last alternative might insert as text part statements other than a statement list or a conditional statement (in particular a block) thereby producing a machine state which does not satisfy the abstract syntax of state. But in this case, in the next step either goto-3 or goto-4, both using the function take-st, will lead to an error. This situation arises, if either a forbidden goto into a block is tried, or if the index list of the label does not localize a statement.

(4.53) goto-4(index) =

```
s-ci:μ(CI;<s-index:index>)
s-c :int-st-1
```

Ref.: int-st-1 4-14(28)

4.6 The Manipulation and Modification of Data

This section involves the definition of the instructions, causing manipulation and modification of data, which model the interpretation of the assignment statement and the evaluation of expressions. This involves, in the main, changes in the value part of the denotation directory DN.

Metavariables:

sel	is-value-sel	value selector
v	is-simple-val	simple value

4.6.1 The assignment statement

The assignment statement is interpreted by the instruction int-assign-st. The interpretation involves the evaluation of a left part list, the evaluation of the right part expression and the assignment (with possible conversion) of the value of the latter to each of the members of the evaluated left part list.

The result of the evaluation of a left part is called a reference and consists of a type component and a value selector component. The value selector component is a compound selector which, when applied to the denotation directory DN, yields a simple value. Hence, one can think of the reference as a sort of address with associated type information. Such references are used whenever data is to be accessed or stored in the denotation directory.

Formally a reference is defined by:

```
is-ref = (<s-type:is-type>,
          <s-value-sel:is-value-sel>)

where: is-value-sel(χ) =
          (∃ n)(χ= s-value∘n & is-n(n)) ∨
          (∃ i,χ-1)(χ= (elem(i))∘χ-1 & is-intg-val(i) & is-value-sel(χ-1))
```

The result of the evaluation of an expression is called an operand and consists of a type component and a value component (which latter is always a simple value in ALGOL 60). The formal definition of the predicate characterizing operands is:

```
is-op = (<s-type:is-type>,
         <s-value:is-simple-val>)
```

The type information becomes essential in the conversion of values to the type specified by a reference by means of which the value is to be assigned to a variable.

(4.54) int-assign-st(t) =

```
convert-assign-list(ref-list,op);
  op:eval-expr(s-rp(t),E);
    ref-list:eval-lp-list(s-lp(t))
```

for : is-assign-st(t)
Ref.: eval-expr 4-25(66)

(4.55) eval-lp-list(lp-list) =

```
is-<>(lp-list) ⟶ PASS:<>
T ⟶
  mk-list(ref,ref-list);
   ref-list:eval-lp-list(tail(lp-list));
         ref:eval-lp(head(lp-list),E)
```

for : is-var-list(lp-list)
Ref.: tail 2-20/2/, head 2-20/2/
Note: The elements of the left part list are evaluated from left to right.

(4.56) mk-list(elem,list) =

PASS:<elem> ^ list

(4.57) eval-lp(lp,env) =

```
is-subscr-var(lp) & is-var-den(final-den_lp) ⟶
  eval-lp-1(s-da(final-den_lp),s-value∘final-n_lp,s-subscr-list(lp),env)
¬is-id(lp) ⟶ error
is-var-den(den_lp) ⟶
  eval-lp-1(s-da(den_lp),s-value∘n_lp,<>,Ω)
is-proc-den(den_lp) ⟶
  eval-lp-1(s-da(n_lp(env)(DN)),s-value∘(n_lp(env)),<>,Ω)
is-name-par-den(den_lp) & is-var∘s-text(den_lp) ⟶
  eval-lp(s-text(den_lp),s-e(den_lp))
T ⟶ error
```

for : is-var(lp)
where: n_{lp} = lp(env), den_{lp} = n_{lp}(DN)
$final\text{-}n_{lp}$ = final-n(s-id(lp)(env),DN)
$final\text{-}den_{lp}$ = $final\text{-}n_{lp}$(DN)
Note : First error if a left part is a switch designator or a label. The second type of error occurs if the identifier is a formal parameter but the text of the corresponding actual parameter is not a variable.

(4.58) eval-lp-1(eda,sel,subscr-list,env) =

```
is-type(eda) & is-<>(subscr-list) ⟶
  PASS:μ_0(<s-type:eda>,
           <s-value-sel:sel>)
is-type(eda) ∨ is-<>(subscr-list) ⟶ error
T ⟶
  eval-lp-1(s-elem(eda),sel-1,tail(subscr-list),env);
    sel-1:pass-value-sel(i,sel);
          i:test-subscript(j,s-lbd(eda),s-ubd(eda));
            j:eval-intg-expr(head(subscr-list),env)
```

for : is-expr-list(subscr-list)
Ref.: head 2-20/2/, tail 2-20/2/
Note: Error: if the subscript list of the left part and the number of dimensions of the array do not coincide.

(4.59) eval-intg-expr(expr,env) =

```
convert(INTG,op);
  op:eval-expr(expr,env)
```

Ref.: eval-expr 4-25(66)

(4.60) test-subscript(i,lbd,ubd) =

```
lbd≤i≤ubd ⟶ PASS:i
T ⟶ error
```

(4.61) value-sel(i,sel) =

```
(elem(i))∘sel
```

(4.62) convert-assign-list(ref-list,op) =

```
(∀ i)(1≤i≤length(ref-list) ⊃
      s-type∘elem(i,ref-list) = s-type∘head(ref-list)) ⟶
  null;
    {convert-assign(elem(i,ref-list),op) | 1≤i≤length(ref-list)}
T ⟶ error
```

Ref.: null 4-7(8), length 2-20/2/
Note: Error: if the types of the left parts are not identical.

(4.63) convert-assign(ref,op) =

```
assign(s-value-sel(ref),v);
  v:convert(s-type(ref),op)
```

(4.64) assign(sel,v) =

s-dn:μ(DN;<sel:v>)

(4.65) convert(type,op) =

type = s-type(op) ⟶ PASS:s-value(op)

is-REAL(type) & is-INTG∘s-type(op) ⟶

PASS:s-value(op)

is-INTG(type) & is-REAL∘s-type(op) ⟶

PASS:(ιi)(is-intg-val(i) & i-1/2≤s-value(op)<i 1/2)

T ⟶ error

Note: Error: if the conversion is attempted between incompatible types.

4.6.2 Evaluation of expressions

Expressions are evaluated by the instruction eval-expr having as arguments some text, which may be an expression or a string (in the case of an actual parameter passed to a procedure), and an environment env, which is the environment to be used for the evaluation. If the evaluation is successful the result is an operand. The evaluation of the two operands of an infix-expression proceeds in arbitrary order.

(4.66) eval-expr(t,env) =

is-cond-expr(t) ⟶

eval-cond-expr(truth,s-then-expr(t),s-else-expr(t),env);
truth:eval-truth(s-decision(t),env)

is-infix-expr(t) ⟶

infix-op(op-1,op-2,s-opr(t));
op-1:eval-expr(s-op-1(t),env),
op-2:eval-expr(s-op-2(t),env)

is-prefix-expr(t) ⟶

prefix-op(op,s-opr(t));
op:eval-expr(s-op(t),env)

is-funct-des(t) ⟶ eval-funct-des(t,env)
is-subscr-var(t) ⟶ eval-var(t,env)
is-id(t) ⟶ eval-id(t,env)
is-const(t) ⟶ PASS:t

T ⟶ error

for : (is-expr ∨ is-string)(t)
Ref.: eval-truth 4-15(34)

cont'd

Note: Error: if the instruction has been used for installing the arguments of a procedure statement and one of these arguments was a string.

(4.67) eval-cond-expr(truth,then-expr,else-expr,env) =

truth ⟶ eval-expr(then-expr,env)

T ⟶ eval-expr(else-expr,env)

for: is-expr(then-expr) & is-expr(else-expr)

(4.68) eval-funct-des(expr,env) =

get-op(ref);
ref:eval-lp-1(s-da•n(DN),s-value•n,<>,Ω);
int-proc-st(expr,env,n);
n:un-name

Ref.: eval-lp-1 4-24(58), int-proc-st 4-9(17), un-name 3-6(11)

Note: The core of this instruction is the instruction corresponding to a procedure statement and the additional instructions that are invoked serve the purpose of saving the value of the function designator after the interpretation of the procedure statement. Therefore, a new unique name is generated with which a reference is constructed and used to temporarily store and access a simple value in the denotation directory.

(4.69) eval-var(var,env) =

get-op(ref);
ref:eval-lp(var, env)

for : is-var(var)
Ref.: eval-lp 4-23(57)

(4.70) get-op(ref) =

is-simple-val(s-value-sel(ref)(DN)) ⟶

PASS:mk-op(s-type(ref),s-value-sel(ref)(DN))

T ⟶ error

Note: Error: if the reference does not yield a value, e.g., there is an exit by means of a goto before the assignment of a value.

(4.71) mk-op(type,v) =

μ_o(<s-type:type>,
<s-value:v>)

(4.72) eval-id(id,env) =

is-var-den(den_{id}) ⟶ eval-var(var_{id},env)
is-proc-den(den_{id}) ⟶ eval-funct-des($funct\text{-}des_{id}$,env)
is-name-par-den(den_{id}) ⟶ eval-expr(s-text(den_{id}),s-e(den_{id}))
T ⟶ error

where: den_{id} = id(env)(DN)

var_{id} = μ_o(<s-id:id>, <s-subscr-list:<>>)

$funct\text{-}des_{id}$ = μ_o(<s-id:id>, <s-subscr-list:<>>)

Note : var_{id} and $funct\text{-}des_{id}$ are used to permit the use of uniform instructions for interpreting simple variables, subscripted variables and identifiers denoting function designators. Error: if the identifier denotes a label or a switch.

(4.73) prefix-op(op,opr) =

is-BOOL($type_1$) & is-NOT(opr) ⟶ PASS:mk-op(BOOL,$\neg v_1$)
is-INTG($type_1$) & is-PLUS(opr) ⟶ PASS:op
is-INTG($type_1$) & is-MINUS(opr) ⟶ PASS:mk-op(INTG,$-v_1$)
is-REAL($type_1$) & (is-PLUS(opr) ∨ is-MINUS(opr)) ⟶
PASS:mk-op(REAL,real-prefix-value(v_1,opr))
T ⟶ error

where: v_1 = s-value(op)
$type_1$ = s-type(op)

Note : Error: if the types of the operand and the operator are incompatible. e.g., is-BOOL($type_1$) & is-PLUS(opr).

(4.74) real-prefix-value(v,opr) =

Note: This function is implementation defined and yields a result satisfying the predicate is-real-val.

(4.75) infix-op(op-1,op-2,opr) =

is-BOOL($type_1$) & is-BOOL($type_2$) & is-bool-infix-opr(opr) ⟶
PASS:mk-op(BOOL,bool-infix-value(v_1,v_2,opr))
is-INTG($type_1$) & is-INTG($type_2$) & is-relat-opr(opr) ⟶
PASS:mk-op(BOOL,intg-relat-value(v_1,v_2,opr))
is-arithm($type_1$) & is-arithm($type_2$) & is-relat-opr(opr) ⟶
PASS:mk-op(BOOL,real-relat-value(v_1,v_2,opr))
is-arithm($type_1$) & is-INTG($type_2$) & is-POWER(opr) ⟶
intg-power-op(op-1,v_2)

cont'd

is-arithm($type_1$) & is-REAL($type_2$) & is-POWER(opr) ⟶
PASS:mk-op(REAL,real-power-value(v_1,v_2))

is-INTG($type_1$) & is-INTG($type_2$) & is-arithm-infix-opr(opr) & ¬is-DIV(opr) ⟶
PASS:mk-op(INTG,intg-arithm-infix-value(v_1,v_2,opr))

is-arithm($type_1$) & is-arithm($type_2$) &
is-arithm-infix-opr(opr) & ¬is-INTGDIV(opr) ⟶
PASS:mk-op(REAL,real-arithm-infix-value(v_1,v_2,opr))

T ⟶ error

for : is-infix-opr(opr)

where: v_1 = s-value(op-1)

v_2 = s-value(op-2)

$type_1$ = s-type(op-1)

$type_2$ = s-type(op-2)

Note : Error: if the types of the operands and the operator of an infix expression are incompatible.

(4.76) bool-infix-value(v-1,v-2,opr) =

is-AND(opr) ⟶ v-1 & v-2
is-OR(opr) ⟶ v-1 ∨ v-2
is-IMPL(opr) ⟶ v-1 ⊃ v-2
is-EQUIV(opr) ⟶ v-1 ≡ v-2

for: is-bool-infix-opr(opr) & is-bool-val(v_1) & is-bool-val(v_2)

(4.77) intg-relat-value(v-1,v-2,opr) =

is-GT(opr) ⟶ v-1 > v-2
is-GE(opr) ⟶ v-1 ≥ v-2
is-EQ(opr) ⟶ v-1 = v-2
is-LE(opr) ⟶ v-1 ≤ v-2
is-LT(opr) ⟶ v-1 < v-2
is-NE(opr) ⟶ v-1 ≠ v-2

for: is-relat-opr(opr) & is-intg-val(v-1) & is-intg-val(v-2)

(4.78) real-relat-value(v-1,v-2,opr) =

for : is-relat-opr(opr) & is-real-val(v-1) & is-real-val(v-2)
Note: This function is implementation defined and yields a result satisfying the predicate is-bool-val.

(4.79) intg-arithm-infix-value(v-1,v-2,opr) =

```
is-ADD(opr)      ⟶ v-1 + v-2
is-SUBTR(opr)    ⟶ v-1 - v-2
is-MULT(opr)     ⟶ v-1 * v-2
is-INTGDIV(opr)  ⟶ sign(v-1/v-2) * entier∘abs(v-1/v-2)
```

for: (is-ADD ∨ is-SUBTR ∨ is-MULT ∨ is-INTGDIV)(opr) &
is-intg-val(v-1) & is-intg-val(v-2)

(4.80) real-arithm-infix-value(v-1,v-2,opr) =

Note: This function is implementation defined and yields a result satisfying the predicate is-real-val.

(4.81) intg-power-op(op,i) =

```
i > 1 ⟶
  infix-op(op-1,op,MULT);
    op-1:intg-power-op(op,i-1)
i = 1  ⟶ PASS:op
v1 = 0 ⟶ error
i = 0  ⟶ PASS:mk-op(s-type(op),1)
i < 0  ⟶
  infix-op(mk-op(REAL,1),op-1,DIV);
    op-1:intg-power-op(op,-i)
```

where: v_1 = s-value(op)

for : is-intg-val(i)

(4.82) real-power-value(v-1,v-2) =

```
v-1 > 0 ⟶ real-arithm-infix-value(v-1,v-2,POWER)
v-1 = 0 & v-2 > 0 ⟶ 0
T ⟶ error
```

5. THE DEFINITION OF A CONCRETE REPRESENTATION OF ABSTRACT PROGRAMS

To define a concrete representation for the programs as specified by the abstract syntax in chapter 2 means to associate with each program character strings (possibly none, see p. 1-1, second paragraph) of finite length.

In this chapter the association of abstract programs with their concrete representations is accomplished by means of a representation system of the type described and defined in section 3.2.1 of /2/. This representation system is said to define the set of concrete ALGOL 60 programs.

5.1 The Generalized Representation System

The central part of the system is a set of conditional replacement schemata which permit the specification of replacement processes leading from abstract programs to their possible concrete representation (cf. 3.2.1 of /2/). The notion of conditional replacement schemata defined in /2/ must, however, be extended to include some additional schemata. In /2/ a conditional replacement schema had the following general form:

$$p(x_1,\ldots,x_m,t_1,\ldots,t_n):\text{NONTERM}[t_1,\ldots,t_n] \Longrightarrow \gamma(x_1,\ldots,x_m,t_1,\ldots,t_n)$$

where $t_1,\ldots,t_n$ are the parameters and $x_1,\ldots,x_m$ are the auxiliary variables of the rule. p is a meta-expression, called the condition of the rule, and denotes a truth-value depending on the free variables $x_1,\ldots,x_m,t_1,\ldots,t_n$. γ denotes a string consisting of non-terminal expressions and terminal expressions in the free variables $x_1,\ldots,x_m,t_1,\ldots,t_n$.

In this chapter the conditional replacement schema will be generalized to have the following general form:

$$p(x_1,\ldots,x_m,t_1,\ldots,t_n): \gamma(t_1,\ldots,t_n) \Longrightarrow \gamma_1(x_1,\ldots,x_m,t_1,\ldots,t_m) \mid \ldots \mid \gamma_k(x_1,\ldots,x_m,t_1,\ldots,t_n)$$

where the left hand side between ':' and ' $\Longrightarrow$ ' may now be a string consisting of at least one non-terminal expression and terminal expressions in the free variables $x_1,\ldots,x_m,t_1,\ldots,t_n$.

A schema of the above type is to be understood in the following sense: For each assignment of specific objects $x_1^\circ,\ldots,x_m^\circ,t_1^\circ,\ldots,t_n^\circ$ as values of the variables $x_1,\ldots,x_m,t_1,\ldots,t_n$, if the condition p is satisfied, then the string $\gamma(t_1^\circ,\ldots,t_n^\circ)$ may be replaced at any of its occurrences in a given string by either one of the strings $\gamma_1(x_1^\circ,\ldots,x_m^\circ,t_1^\circ,\ldots,t_n^\circ)$ or ... or $\gamma_k(x_1^\circ,\ldots,x_m^\circ,t_1^\circ,\ldots,t_n^\circ)$.

Non-terminal names can now be words consisting of capital letters optionally followed by a hyphon followed by a digit.

A replacement process is a sequence $\gamma_0, \gamma_1, \ldots, \gamma_k$, where γ_0 is the non-terminal H[t°] and γ_{i+1} is obtained from γ_i (for $0 \leq i < k$) by the application of one of the conditional replacement schemata of $\mathcal{R}$. The last string γ_k of the sequence consists solely of terminal symbols and constitutes a concrete representation of t° .

5.2 The Representation System for Abstract Programs

The representation system for ALGOL 60 is defined as the quintuple

$$\langle \mathcal{A}, \mathcal{T}, \mathcal{N}, \underline{H}, \mathcal{R} \rangle$$

where $\mathcal{A}$ is the abstract syntax of chapter 2 (cf. definitions (2.1) to (2.52)), $\mathcal{T}$ is the set of basic symbols of ALGOL 60 (cf. /1/, section 2), $\mathcal{N}$ is the set of non-terminal names, $\underline{H}$ is the unique non-terminal name called the head of the system and $\mathcal{R}$ is the set of conditional replacement schemata.

The set of non-terminal names $\mathcal{N}$ is defined by means of a tabular listing of its elements. Each member of the set is accompanied by a suggested reading. The first non-terminal name is the head of the system, hence:

H = PROGRAM

(N1)	APAR	actual parameter
(N2)	APARL	actual parameter list
(N3)	BEGIN	begin or comment
(N4)	BLOCK	block
(N5)	BODY	body of a procedure
(N6)	BPL	bound pair list
(N7)	COMTSTR	comment string
(N8)	COMTSYM	comment symbol
(N9)	CONDST	conditional statement
(N10)	CONST	constant
(N11)	DECL	declaration part
(N12)	END	end or comment
(N13)	EXPR	expression
(N14)	EXPRL	expression list
(N15)	FOREL	for list element
(N16)	FORL	for list element list
(N17)	FPARL	formal parameter list
(N18)	IDL	identifier list
(N19)	INFL	left hand expression of an infix expression
(N20)	INFR	right hand expression of an infix expression
(N21)	INTGVAL	integer value
(N22)	LAB	label
(N23)	LABSET	label set
(N24)	LETSTR	letterstring
(N25)	LETSYM	lettersymbol
(N26)	LPL	left part list
(N27)	LST	labeled statement
(N28)	PARDEL	parameter delimiter
(N29)	PROCST	procedure statement
(N30)	PROGRAM	program
(N31)	REALVAL	real value
(N32)	SCOPE	scope
(N33)	SEMIC	semicolon or comment
(N34)	SPEC	specification
(N35)	SPECPT	specification part
(N36)	ST	statement
(N37)	STL	statement list
(N38)	TYPE	type
(N39)	VALUE	value

The set of conditional replacement schemata R is listed in section 5.2.2

5.2.1 Auxiliary functions and predicates applying to arguments of the conditional replacement schemata

The conditions and strings occurring in conditional replacement schemata may contain in their argument positions arbitrary meta expressions involving functions and predicates applicable to objects and their components. These are either general functions and predicates defined for objects (in /2/ or in the present report) or predicates defined by the abstract syntax of text or one of the predicates and functions defined in this section.

(5.1) is-progr-block(t) =

$$\begin{aligned}
&\text{is-block}(t)\ \&\\
&(\forall\ \text{name})(\text{is-name}(\text{name})\ \&\ \neg(\text{name}(dp_t) = \Omega\ \&\ \text{name}(DP_o) = \Omega) \supset\\
&\qquad \text{is-label-decl}\circ\text{name}(dp_t)\ \vee\ \text{name}(dp_t) = \text{name}(DP_o))\ \&\\
&\text{length}(\text{st-list}_t) = 1\ \&\\
&(\text{is-block}\circ\text{head}(\text{st-list}_t)\ \vee\ \text{is-comp-st}\circ\text{head}(\text{st-list}_t))
\end{aligned}$$

for : is-program(t)

where: dp_t = s-decl-pt(t)

st-list_t = s-st-list(t)

Note : DP_o is the constant object called the standard declaration part. It satisfies the predicate is-decl-pt and, in addition, it satisfies the condition that all identifiers declared in it are identifiers of standard functions and that their corresponding declarations are of the appropriate (implementation defined) kind, as specified in the ALGOL 60 Revised Report (cf. /1/, 3.2.4).

(5.2) place-labels(t) =

$$\begin{aligned}
&\mu(t;\{\langle \text{s-lab}\circ(\text{comp-elem}(\text{indl},\text{s-st-list}(t))) : \text{labs}\rangle \mid \text{is-index-list}(\text{indl})\ \&\\
&\qquad \text{labs} = \{\text{lab} \mid \text{is-name}(\text{lab})\ \&\ \text{indl} = \text{lab}\circ\text{s-decl-pt}(t)\}\ \&\\
&\qquad \text{labs} \neq \{\}\})
\end{aligned}$$

for : is-block(t)

Note: This function places all labels, declared in the declaration part of t, before the statements indicated by the index list associated with the label in the declaration part.

(5.3) comp-elem(indl,t) =

$$\begin{aligned}
&\text{length}(\text{indl}) = 1 \longrightarrow \text{sel}_1\\
&\text{length}(\text{indl}) > 1 \longrightarrow (\text{comp-elem}(\text{tail}(\text{indl}),\text{sel}_1(t)))\circ\text{sel}_1
\end{aligned}$$

for : (is-block $\vee$ is-comp-st)(t) & is-index-list(indl)

where: sel_1 = mk-sel(head(indl),t)

(5.4) mk-sel(ind,t) =

$$\begin{array}{ll} \text{is-}\Omega\circ\text{sel}_1(t) & \longrightarrow \text{ error} \\ T & \longrightarrow \text{'sel}_1 \end{array}$$

for : is-index(ind) & is-st(t)
where: sel_1 = mk-sel-1(ind)

(5.5) mk-sel-1(index) =

$$\begin{array}{ll} \text{is-T(index)} & \longrightarrow \text{s-then-st} \\ \text{is-F(index)} & \longrightarrow \text{s-else-st} \\ \text{is-FOR(index)} & \longrightarrow \text{s-st} \\ \text{is-intg-val(index) \& index} > 0 & \longrightarrow \text{elem(index)} \\ T & \longrightarrow \text{error} \end{array}$$

for: is-index(index)

(5.6) is-l-[pred](lt) =

$$(\exists t,K)(\text{is-[pred]}(t) \;\&\; \text{lt} = \mu(t;\{\langle \text{s-lab}\circ\chi : \text{s-lab}\circ\chi(\text{lt})\rangle \mid \chi\in K\}) \;\&\; (\forall\chi)(\chi\in K \supset \text{is-name-set}\circ\text{s-lab}\circ\chi\,(\text{lt})))$$

Note: This definition schema defines predicates for abstract text, as modified by the insertion of labels, from the predicates of the abstract syntax of chapter 2.

(5.7) mk-list(id) =

$$\mu_0(\langle \text{elem(1)} : \text{id}\rangle)$$

for: is-id(id)

(5.8) delete-val-par(spec-pt,par-list) =

$$\{\text{id} \mid \text{id(spec-pt)} = \Omega \;\&\; (\exists\, \text{i})(\text{elem(i)(par-list)} = \text{id})\}$$

for : is-spec-pt(spec-pt) & is-id-list(par-list)
Note: This function yields the set of by name parameters from the parameter list of a procedure declaration.

(5.9) is-fictitious-block(t) =

$$\text{is-block(t)} \;\&\; \text{s-decl-pt(t)} \neq \Omega \;\&\; \text{length}\circ\text{s-st-list(t)} = 1 \;\&\; \text{s-decl-pt}\circ\text{erase-label-decls(t)} = \Omega$$

for : is-block(t)
Note: This predicate characterizes bodies of procedures that are blocks whose declaration parts contain only label declarations.

(5.10) erase-label-decls(t) =

$$\delta(t;\{lab \mid \text{is-name}(lab) \ \&\ lab \circ \text{is-label-decl} \circ \text{s-decl-pt}(t)\})$$

for : is-block(t)

Note: This function erases all label declarations local to a given labeled block.

(5.11) prior(opr) =

$$\begin{aligned}
(\text{is-PLUS} \vee \text{is-MINUS} \vee \text{is-ADD} \vee \text{is-SUBTR})(opr) &\longrightarrow 1\\
(\text{is-MULT} \vee \text{is-INDIV} \vee \text{is-DIV})(opr) &\longrightarrow 2\\
\text{is-POWER}(opr) &\longrightarrow 3\\
\text{is-EQUIV}(opr) &\longrightarrow 4\\
\text{is-IMPL}(opr) &\longrightarrow 5\\
\text{is-OR}(opr) &\longrightarrow 6\\
\text{is-AND}(opr) &\longrightarrow 7\\
\text{is-NOT}(opr) &\longrightarrow 8\\
(\text{is-LT} \vee \text{is-LE} \vee \text{is-EQ} \vee \text{is-GE} \vee \text{is-GT} \vee \text{is-NE})(opr) &\longrightarrow 9
\end{aligned}$$

for : (is-prefix-opr ∨ is-infix-opr)(opr)

Note: This function defines a depth of priority of operators. This depth of priority is utilized in going from abstract expressions to their concrete representations.

(5.12) rep-id(id) =

Note: The domain of this one-to-one function is the infinite set of abstract identifiers characterized by is-id and its range is the corresponding set of concrete ALGOL 60 identifiers as defined by /1/.

(5.13) rep-code(t) =

for : is-code(t)

Note: This function is implementation defined.

(5.14) rep-bas-symb(t) =

for : is-basic-symbol(t)

Note: The domain of this one-to-one function is the set of abstract objects characterized by is-basic-symbol and its range is the corresponding set of concrete ALGOL 60 basic symbols as specified in /1/, section 2.

(5.15) rep-opr(opr) =

```
(is-PLUS ∨ is-ADD)(opr)      ⟶ +
(is-MINUS ∨ is-SUBTR)(opr)   ⟶ -
          is-MULT(opr)       ⟶ *
          is-INTDIV(opr)     ⟶ ÷
          is-POWER(opr)      ⟶ ↑
          is-DIV(opr)        ⟶ /
          is-OR(opr)         ⟶ ∨
          is-AND(opr)        ⟶ ∧
          is-IMPL(opr)       ⟶ ⊃
          is-EQUIV(opr)      ⟶ ≡
          is-NOT(opr)        ⟶ ¬
          is-LT(opr)         ⟶ <
          is-LE(opr)         ⟶ ≤
          is-EQ(opr)         ⟶ =
          is-GE(opr)         ⟶ ≥
          is-GT(opr)         ⟶ >
          is-NE(opr)         ⟶ ≠
```

for: (is-infix-opr ∨ is-prefix-opr)(opr)

(5.16) is-proper-expr =

```
is-arithm-expr ∨ is-bool-expr ∨ is-des-expr
```

(5.17) is-arithm-expr(t) =

```
(is-bool-expr∘s-decision(t) & is-arithm-expr∘s-then-expr(t) &
 is-arithm-expr∘s-else-expr(t)) ∨
(is-arithm-infix-opr∘s-opr(t) &  is-arithm-expr∘s-op-1(t) &
 is-arithm-expr∘s-op-2(t)) ∨
(is-prefix-expr(t) & ¬is-NOT∘s-opr(t) & is-arithm-expr∘s-op(t)) ∨
(is-funct-ref ∨ is-var ∨ (is-const & ¬is-bool-const))(t)
```

(5.18) is-des-expr(t) =

```
(is-bool-expr∘s-decision(t) & is-des-expr∘s-then-expr(t) &
 is-des-expr∘s-else-expr(t)) ∨
(is-subscr-var(t) & length∘s-subscr-list(t) = 1) ∨
(is-intg-const(t) & s-value(t) ≥ 0) ∨ is-id(t)
```

(5.19) is-bool-expr(t) =

```
(is-bool-expr∘s-decision(t) & is-bool-expr∘s-then-expr(t) &
 is-bool-expr∘s-else-expr(t)) ∨
((is-bool-infix-opr∘s-opr(t) & is-bool-expr∘s-op-1(t) &
 is-bool-expr∘s-op-2(t)) ∨ (is-relat-infix-opr∘s-opr(t) &
 is-arithm-expr∘s-op-1(t) &
 is-arithm-expr∘s-op-2(t))) ∨
(is-NOT∘s-opr(t) & is-bool-expr∘s-op(t)) ∨
(is-funct-ref ∨ is-var ∨ is-bool-const)(t)
```

5.2.2 The conditional replacement schemata of the representation system

In this section the definition of the representation system is completed by listing the elements of the set $\mathcal{R}$ of conditional replacement schemata. The parentheses, comma, semicolon, colon and colon-equal sign are written especially heavy to indicate that they are terminal symbols standing for themselves.

(5.20) is-progr-block(t):PROGRAM[t] ⟹

LST[head∘s-st-list∘place-labels(t)]

(5.21) LST[t] ⟶ LABSET[s-lab(t)]ST[δ(t;s-lab)]

(5.22) t = Ω ∨ t = {} : LABSET[t] ⟶ λ

x ∈ t : LABSET[t] ⟶ LAB[x] ⁚ LABSET[t - {x}]

Note: λ denotes the empty string.

(5.23) is-id(t):LAB[t] ⟶ rep-id(t)

t = elem(i):LAB[t] ⟶ INTGVAL[i]

(5.24) is-block(t):ST[t] ⟹ BLOCK[erase-label-decls∘place-labels(t)]

is-1-comp-st(t):ST[t] ⟹

BEGIN STL[t]END

is-proc-st(t):ST[t] ⟹ PROCST[t]

is-1-for-st(t):ST[t] ⟹

for EXPR[s-contr-var(t)] ⁚= FORL[s-for-list(t)] do LST[s-st(t)]

is-dummy-st(t):ST[t] ⟹ λ

is-1-cond-st(t) & is-bool-expr∘s-decision(t):ST[t] ⟹ CONDST[t]

is-goto-st(t) & is-des-expr∘s-label(t):ST[t] ⟹ goto EXPR[s-label(t)]

is-assign-st(t) & s-lp(t) ≠ <>:ST[t] ⟹ LPL[s-lp(t)]EXPR[s-rp(t)]

(5.25) s-decl-pt(t) $\neq \Omega$:BLOCK[t] $\Rightarrow$
BEGIN DECL[s-decl-pt(t)]STL[s-st-list(t)] END

Note: An empty declarátion part is forbidden, since it would lead to the same concrete program as the compound statement corresponding to t.

(5.26) t = Ω:DECL[t] $\Rightarrow$ λ

is-var-decl∘id(t) & is-type∘s-da∘id(t):DECL[t] $\Longrightarrow$ λ
SCOPE[s-scope∘id(t)]TYPE[s-da∘id(t)]IDL[mk-list(id)]SEMIC DECL[δ(t;id)]

is-var-decl∘id(t) & is-array∘s-da∘id(t):DECL[t] $\Longrightarrow$
SCOPE[s-scope∘id(t)] TYPE[get-type∘s-da∘id(t)] array
IDL[mk-list(id)] [BPL[s-da∘id(t)]]SEMIC DECL[δ(t;id)]

is-proc-decl∘id(t):DECL[t] $\Longrightarrow$
TYPE[$type_{id}$] procedure rep-id(id)FPARL[$par\text{-}list_{id}$] SEMIC
VALUE[$spec\text{-}pt_{id}$]SPECPT[$spec\text{-}pt_{id}$,delete-val-par($spec\text{-}pt_{id}$,$par\text{-}list_{id}$)]
BODY[$body_{id}$] SEMIC DECL[δ(t;id)]

is-switch-decl∘id(t) & ($\forall$i)(1≤i≤length(id(t)) ⊃
is-des-expr∘elem(i,id(t))):DECL[t] $\Longrightarrow$
switch rep-id(id):■ EXPRL[id(t)]SEMIC DECL[δ(t;id)]

where: $type_{id}$ = s-type∘id(t)

$par\text{-}list_{id}$ = s-par-list∘id(t)

$spec\text{-}pt_{id}$ = s-spec-pt∘id(t)

$body_{id}$ = s-body∘id(t)

Ref. : delete-val-par 5-5(8)

(5.27) t = Ω:SCOPE[t] $\Longrightarrow$ λ
t = OWN:SCOPE[t] $\longrightarrow$ own

(5.28) t = Ω:TYPE[t] $\Longrightarrow$ λ
t = INTG:TYPE[t] $\Longrightarrow$ integer
t = REAL:TYPE[t] $\Longrightarrow$ real
t = BOOL:TYPE[t] $\Longrightarrow$ Boolean

(5.29) length(t) = 1:IDL[t] $\Longrightarrow$ rep-id∘head(t)
length(t) > 1:IDL[t] $\Longrightarrow$ rep-id∘head(t), IDL[tail(t)]

(5.30) length(t) = 0:FPARL[t] $\Longrightarrow$ λ
length(t) > 0:FPARL[t] $\Longrightarrow$ (FPARL-1[t])

(5.31) length(t) = 1:FPARL-1[t] $\Longrightarrow$ rep-id∘head(t)
length(t) > 1:FPARL-1[t] $\Longrightarrow$ rep-id∘head(t) PARDEL FPARL-1[tail(t)]

(5.32) PARDEL $\Longrightarrow$, |) LETSTR : (

(5.33) LETSTR $\Longrightarrow$ LETSYM | LETSYM LETSTR

(5.34) LETSYM $\Longrightarrow$ a | b | c | d | e | f | g | h | i | j | k | l | m
n | o | p | q | r | s | t | u | v | w | x | y | z
A | B | C | D | E | F | G | H | I | J | K | L | M
N | O | P | Q | R | S | T | U | V | W | X | Y | Z

(5.35) is-type∘s-elem(t) &
is-arithm-expr∘s-lbd(t) & is-arithm-expr∘s-ubd(t):BPL[t] $\Longrightarrow$
EXPR[s-lbd(t)]:EXPR[s-ubd(t)]

$\neg$is-type∘s-elem(t) &
is-arithm-expr∘s-lbd(t) & is-arithm-expr∘s-ubd(t):BPL[t] $\Longrightarrow$
EXPR[s-lbd(t)] : EXPR[s-ubd(t)] , BPL[s-elem(t)]

(5.36) t $=\Omega$:VALUE[t] $\Longrightarrow$ λ
t $\neq\Omega$:VALUE[t] $\Longrightarrow$ value VALUE-1[t] SEMIC

(5.37) t $=\Omega$:VALUE-1[t] $\Longrightarrow$ λ
id(t) $\neq\Omega$:VALUE-1[t] $\Longrightarrow$ rep-id(id) VALUE-2[δ(t;id)]

(5.38) t $=\Omega$:VALUE-2[t] $\Longrightarrow$ λ
id(t) $\neq\Omega$:VALUE-2[t] $\Longrightarrow$, VALUE-1[t]

(5.39) $t_3 \subseteq t_2$: SPECPT[t_1,t_2] $\Longrightarrow$ SPECPT-1[t_1,t_3]

(5.40) $t_1 = \Omega$ & $t_2 = \{\ \}$: SPECPT-1[t_1,t_2] $\Longrightarrow$ λ
id(t_1) $\neq\Omega$: SPECPT-1[t_1,t_2] $\Longrightarrow$
SPEC[id(t_1),mk-list(id)]SPECPT-2[δ(t_1;id),t_2]
id $\in t_2$: SPECPT-1[t_1,t_2] $\Longrightarrow$
SPEC-1[mk-list(id)]SPECPT-2[t_1,t_2-{id}]

(5.41) $t_1 = \Omega$ & $t_2 = \{\ \}$: SPECPT-2[t_1,t_2] $\Longrightarrow$ SEMIC
$t_1 \neq \Omega \vee t_2 \neq \{\ \}$: SPECPT-2[t_1,t_2] $\Longrightarrow$, SPECPT-1[t_1,t_2]

(5.42) is-type(t_1) : SPEC[t_1,t_2] $\Longrightarrow$ TYPE[t_1]IDL[t_2]
is-LABEL(t_1) : SPEC[t_1,t_2] $\Longrightarrow$ label IDL[t_2]
is-type-array(t_1) : SPEC[t_1,t_2] $\Longrightarrow$
TYPE[s-array-type(t_1)] array IDL[t_2]

(5.43) SPEC-1[t] ⟹ string IDL[t] | integer IDL[t] | real IDL[t] |
Boolean IDL[t] | array IDL[t] | integer array IDL[t] |
real array IDL[t] | Boolean array IDL[t] | label IDL[t] |
switch IDL[t] | procedure IDL[t] |
real procedure IDL[t] | integer procedure IDL[t] |
Boolean procedure IDL[t]

(5.44) is-code(t):BODY[t] ⟹ rep-code(t)

is-fictitious-block(t):BODY[t] ⟹
LST[head∘s-st-list∘place-labels(t)]

is-st & ¬is-fictitious-block(t):BODY[t] ⟹ ST[t]

(5.45) SCOPE[t_1]TYPE[t_2]array IDL[t_3] [BPL[t_4]] SEMIC
SCOPE[t_1]TYPE[t_2]array IDL[t_5] [BPL[t_6]] ⟹
SCOPE[t_1]TYPE[t_2]array IDL[t_3] [BPL[t_4]] ,
IDL[t_5] [BPL[t_6]]

Note: This and the next six rules permit the different types of combinations of identifiers declared with identical attributes which are permitted in /1/.

(5.46) IDL[t_1] [BPL[t_2]] , IDL[t_3] [BPL[t_2]] ⟹
IDL[$t_1 \cap t_3$] [BPL[t_2]]

(5.47) SCOPE[t_1]TYPE[t_2]IDL[t_3] SEMIC SCOPE[t_1]TYPE[t_2]IDL[t_4] ⟹
SCOPE[t_1]TYPE[t_2]IDL[$t_3 \cap t_4$]

(5.48) SPEC[t_1,t_2] , SPEC[t_1,t_3] ⟹ SPEC[t_1,$t_2 \cap t_3$]

(5.49) SPEC-1[t_1] , SPEC-1[t_2] ⟹ SPEC-1[$t_1 \cap t_2$]

(5.50) SPEC[t_1,t_2] , SPEC-1[t_3] ⟹ SPEC[t_1,$t_2 \cap t_3$]

(5.51) SPEC-1[t_1] , SPEC[t_2,t_3] ⟹ SPEC[t_2,$t_1 \cap t_3$]

(5.52) is-REAL(t):TYPE[t]array IDL[t_1] ⟹ array IDL[t_1]

Note: This is the default schema for real declared arrays.

(5.53) length(t) = 1:STL[t] ⟹ LST[head(t)]
length(t) > 1:STL[t] ⟹ LST[head(t)] SEMIC STL[tail(t)]

(5.54) is-<>∘s-arg-list(t):PROCST[t] ⟹ rep-id∘s-id(t)
¬is-<>∘s-arg-list(t):PROCST[t] ⟹
rep-id∘s-id(t) (APARL[s-arg-list(t)])

(5.55) length(t) = 1:APARL[t] ⟹ APAR[head(t)]
length(t) > 1:APARL[t] ⟹ APAR[head(t)]PARDEL APARL[tail(t)]

(5.56) is-string(t):APAR[t] ⟹ 'STRING[t]'
is-expr(t):APAR[t] ⟹ EXPR[t]

(5.57) length(t) = 1:STRING[t] ⟹ STRING-ELEM[head(t)]
length(t) > 1:STRING[t] ⟹ STRING-ELEM[head(t)]STRING[tail(t)]

(5.58) is-basic-symbol(t):STRING-ELEM[t] ⟹ rep-bas-symb(t)
is-string(t):STRING-ELEM[t] ⟹ 'STRING[t]'

(5.59) length(t) = 1:FORL[t] ⟹ FOREL[head(t)]
length(t) > 1:FORL[t] ⟹ FOREL[head(t)], FORL[tail(t)]

(5.60) is-arithm-expr(t):FOREL[t] ⟹ EXPR[t]

is-step-until-elem(t) & is-arithm-expr∘s-init-expr(t) &
is-arithm-expr∘s-step-expr(t) & is-arithm-expr∘s-until-expr(t):FOREL[t] ⟹
EXPR[s-init-expr(t)]step EXPR[s-step-expr(t)]until EXPR[s-until-expr(t)]

is-while-elem(t) & is-arithm-expr∘s-init-expr(t) &
is-bool-expr∘s-while-expr(t):FOREL[t] ⟹
EXPR[s-init-expr(t)]while EXPR[s-while-expr(t)]

(5.61) is-1-dummy-st∘s-else-st(t) & (¬is-1-cond-st ∨ ¬is-1-for-st)∘s-then-st(t):
CONDST[t] ⟹
if EXPR[s-decision(t)]then LST[s-then-st(t)]else LST[s-else-st(t)]

is-1-dummy-st∘s-else-st(t) & ¬is-1-cond-st∘s-then-st(t):CONDST[t] ⟹
if EXPR[s-decision(t)]then LST[s-then-st(t)]

(¬is-1-cond-st ∨ ¬is-1-for-st)∘s-then-st(t) & is-1-cond-st∘s-else-st(t):
CONDST[t] ⟹
if EXPR[s-decision(t)]then LST[s-then-st(t)]
else BEGIN LST[s-else-st(t)]END

(¬is-1-cond-st ∨ ¬is-1-for-st)∘s-then-st(t) & ¬is-1-cond-st∘s-else-st(t):
CONDST[t] ⟹
if EXPR[s-decision(t)]then LST[s-then-st(t)]
else LST[s-else-st(t)]

(5.62) t = <>:LPL[t] ⟹ λ

t ≠ <> & (is-subscr-var∘head(t) ∨ is-id∘head(t)):LPL[t] ⟹
EXPR[head(t)] := LPL[tail(t)]

(5.63) length(t) = 1:EXPRL[t] ⟹ EXPR[head(t)]
length(t) > 1:EXPRL[t] ⟹ EXPR[head(t)], EXPRL[tail(t)]

(5.64) (is-cond-expr & is-proper-expr)(t) & is-cond-expr∘s-then-expr(t):EXPR[t] ⟹
if EXPR[s-decision(t)] then (EXPR[s-then-expr(t)])
else EXPR[s-else-expr(t)]

(is-cond-expr & is-proper-expr)(t) & ¬is-cond-expr∘s-then-expr(t):EXPR[t] ⟹
if EXPR[s-decision(t)] then EXPR[s-then-expr(t)]
else EXPR[s-else-expr(t)]

(is-infix-expr & (is-arithm-expr ∨ is-bool-expr))(t):EXPR[t] ⟹
INFL[s-op-1(t),prior∘s-opr(t)]rep-opr∘s-opr(t)INFR[s-op-2(t),prior∘s-opr(t)]

(is-prefix-expr & (is-arithm-expr ∨ is-bool-expr))(t):EXPR[t] ⟹
rep-opr∘s-opr(t)INFR[s-op(t),prior∘s-opr(t)]

is-funct-ref(t) & s-arg-list(t) ≠ <>:EXPR[t] ⟹
PROCST[t]

is-subscr-var(t) & s-subscr-list(t) ≠ <> & (∀i) (1≤i≤length∘s-subscr-list(t) ⊃
is-arithm-expr∘elem(i,s-subscr-list(t)):EXPR[t] ⟹
rep-id∘s-id(t)[EXPRL[s-subscr-list(t)]]

is-id(t):EXPR[t] ⟹ rep-id(t)

is-const(t):EXPR[t] ⟹ CONST[t]

(5.65) is-cond-expr(t) ∨ (s-opr(t) ≠ Ω & prior∘s-opr(t)<i):
INFL[t,i] ⟹ (EXPR[t])

¬is-cond-expr(t) & (s-opr(t) = Ω ∨ prior∘s-opr(t)≥i):
INFL[t,i] ⟹ EXPR[t]

(5.66) is-cond-expr(t) ∨ (s-opr(t) ≠ Ω & prior∘s-opr(t)≤i):
INFR[t,i] ⟹ (EXPR[t])

¬is-cond-expr(t) & (s-opr(t) = Ω ∨ prior∘s-opr(t)>i):
INFR[t,i] ⟹ EXPR[t]

(5.67) s-value(t) = T:CONST[t] ⟹ true
s-value(t) = F:CONST[t] ⟹ false
is-intg-const(t):CONST[t] ⟹ INTGVAL[s-value(t)]
is-real-const(t):CONST[t] ⟹ REALVAL[s-value(t)]

(5.68) v≥0:INTGVAL[v] ⟹

Note: This non-terminal can be replaced by any elements of the set of possible representations of the integer value v as specified in /1/.

(5.69) v≥0:REALVAL[v] ⟹

Note: This non-terminal can be replaced by any elements of the set of possible representations of the real value v as specified in /1/.

(5.70) EXPR[t] ⟹ (EXPR[t])

Note: This schema is the formal analogue of the statement in /1/ that any expression can be enclosed in additional parentheses.

(5.71) SEMIC ⟹ ; | SEMIC comment COMTSTR ;

Note: This and the following six schemas permit the insertion of comments in the program text as specified by /1/.

(5.72) BEGIN ⟹ begin | BEGIN comment COMTSTR ;

(5.73) END ⟹ end | end COMTSTR-1

(5.74) COMTSTR ⟹ COMTSYM | COMTSYM COMTSTR

(5.75) COMTSYM ⟹

Note: Any basic symbol (as defined by /1/) not equal to ; .

(5.76) COMTSTR-1 ⟹ COMTSYM-1 | COMTSYM-1 COMTSTR-1

(5.77) COMTSYM-1 ⟹

Note: Any basic symbol not equal to end or ; or else.