1 Regnecentralen

Algol Bulletin Supplement no. 2 ABS 2

~~Note: This is a draft report, not endorsed by the authors.~~

1.3.1960

REPORT ON THE ALGORITHMIC LANGUAGE ALGOL 60.

Dedicated to the memory of

WILLIAM TURANSKI

by

J.W.Backus, F.L.Bauer, J.Green, C.Katz, J.McCarthy, P.Naur, A.J.Perlis, H.Rutishauser, K.Samelson, B.Vauquois, J.H.Wegstein, A.van Wijngaarden, M.Woodger

edited by
Peter Naur

INTRODUCTION

NB

Background

After the publication[1,2] of a preliminary report on the algorithmic language ALGOL, as prepared at a conference in Zurich in 1958, much interest in the ALGOL language developed.

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an ALGOL Bulletin, edited by Peter Naur, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal and informal ones, were held. These meetings revealed some misunderstandings as to the intent of the group which was primarily responsible for the formulation of the language, but at the same time made it clear that there exists a wide appreciation of the effort involved. As a result of the discussions it was decided to hold an international meeting in January 1960 for improving

NB

[1] Preliminary report - International Algebraic Language, Comm.Assoc. Comp.Mach. 1, No. 12 (1958), 8.

[2] Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages and the GAMM Committee on Programming, edited by A. J. Perlis and K. Samelson, Numerische Mathematik Bd. 1, S. 41 - 60 (1959).

the ALGOL language and preparing a final report. At a European ALGOL Conference in Paris in November 1959 which was attended by about ~~forty~~ fifty people, seven European representatives were selected to attend the January 1960 Conference, and they represent the following organisations: Association Francaise de Calcul, British Computer Society, Gesellschaft fur Angewandte Mathematik und Mechanik, and

NB

The seven representatives held a final preparatory meeting at Mainz in December 1959.

Meanwhile, in the United States, anyone who wished to suggest changes or corrections to ALGOL was requested to send his comments to the ACM Communications where they were published. These comments then became the basis of consideration for changes in the ALGOL language. Both the SHARE and USE organisations established ALGOL working groups and both organisations were represented on the ACM Committee on Programming Languages. The ACM Committee met in Washington in November 1959 and considered all comments on ALGOL that had been sent to the ACM Communications. Also, seven representatives were selected to attend the January 1960 international conference. These seven representatives held a final preparatory meeting in Boston in December 1959.

January 1960 Conference

The thirteen representatives[1], ~~coming~~ from Denmark, England, France, Germany, Holland, Switzerland, and the United States, conferred in Paris from January 11 to 16, 1960.

Prior to this meeting a completely new draft report was worked out from the preliminary report and the recommendations of the preparatory meetings by Peter Naur and the conference adopted this new form as the basis for its report. The Conference then proceeded to work for agreement on each item of the report. The present report represents the union of the Committee's concepts and the intersection of its agreements.

As with the preliminary ALGOL report, three different levels of language are recognized, namely a Reference Language, a Publication Language and several Hardware Representations.

Reference Language.

1. It is the working language of the committee.
2. It is the defining language.
3. ~~It has only one unique set of characters.~~
   ~~(This is in conflict with section 2.1. I would like to remove it. PN)~~
4. 3 The characters are determined by ~~case~~ ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.

NB

Sam. Bauer
No. com.

[1] William Turanski of the American group was killed by an automobile just prior to the January 1960 Conference.

4 5. It is the basic reference and guide for compiler builders.
5 6. It is the guide for all hardware representations.
6 7. It is the guide for transliterating from publication language to any locally appropriate hardware representations.
7 8. The main publications of the ALGOL language itself will use the reference representation.

Publication Language.

NB

1. ~~The description of this~~ A publication language ~~is in the form of permissible~~ ~~shows~~ admits variations of the reference language (e.g., subscripts, spaces, exponents, Greek letters) according to usage of printing and handwriting. NB
2. It is used for stating and communicating pro~~blems.~~ cesses
3. The characters to be used may be different in different countries, but univocal correspondence with reference representation must be secured.

Hardware Representations.

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.
2. Each one of these uses the character set of a particular computer and is the language accepted by a translator for that computer.
3. Each one of these must be accompanied by a special set of rules for transliterating from Publication or Reference language.

For transliteration between the reference language and a language suitable for publications, among others, the following rules are recommended.

Reference language	Publication language
	~~Brackets replaced by~~
Subscript bracket [ ]	lowering of the line between the brackets. and removal of brackets NB
Exponentation ↑	Raising of the exponent the
Parentheses ( )	Any form of parentheses, brackets braces.
Basis of ten $_{10}$	Raising of the ten and of the following integral number, inserting of the intended multiplication sign.

Wovon man nicht sprechen kann,
darüber muss man schweigen
Ludwig Wittgenstein

DESCRIPTION OF THE REFERENCE LANGUAGE.

1. STRUCTURE OF THE LANGUAGE.

NB

As stated in the introduction, the algorithmic language has three different kinds of representations - reference, hardware, and publication - and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols - and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language - explicit formulae - called assignment statements. / NB

To show the flow of computational processes, certain nonarithmetic statements and statement clauses are added which may describe e.g., alternatives, or iterative ~~recursive~~ repetitions of computing statements. Since it is necessary for the function of these statements that one statement refers to another, statements may be provided with labels. Sequences of statements may be combined into compound statements by insertion of statement brackets. / NB

Statements instructions are supported by declarations which are not themselves computing rules, but inform the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers or even the set of rules defining a function. A Each. declaration is attached to and valid for one compound statement. A compound statement which includes declarations is called a block.

A program is a self-contained compound statement, i.e. a compound statement which is not contained within another compound statement and which makes no use of other compound statements not contained within it.

In the sequel the syntax and semantics of the language will be given.[1]

1.1. FORMALISM FOR SYNTACTIC DESCRIPTION.

The syntax will be described with the aid of metalinguistic formulae. R footnote 7

Their interpretation is best explained by an example:

```
    <ab> ::= ( | [ | <ab>( | <ab><d>
```

Sequences of characters enclosed in the bracket < > represent metalinguistic variables whose values are strings of symbols. The marks ::= and | (the latter with the meaning of or) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the strings denoted.

footnote

sequences

7 → Backus reference

Thus the formula above gives a recursive rule for the formation of values of the variable <ab>. It indicates that <ab> may have the value (or [or that given some legitimate value of <ab>, another may be formed by following it with the character (or by following it with some value of the variable <d>. If the values of <d> are the decimal digits, some values of <ab> are:

```
    [(((1(37(
    (12345(
    (((
    [86
```

NB

In the text explaining the semantics any object which is denoted by a designation, A say, which has been defined syntactically as <A>, will be identical with <A>.

In order to facilitate the study some formulae have been given in more than one place.

Definition:

<empty> ::= (i.e. the null string of symbols)

Side 6

2. BASIC SYMBOLS, IDENTIFIERS, NUMBERS, AND STRINGS. BASIC CONCEPTS.

The reference language is built up from the following basic symbols:

<basic symbol> ::= <letter>|<digit>|<logical value>|<delimiter>

2.1. LETTERS.

NB

<letter> ::= 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

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (i.e. character not coinciding with any digit, logical value or delimiter).

footnote

Letters do not have individual meaning. They are used for forming identifiers and strings. (cf. sections 2.4. Identif., 3.2 Function ~~values~~ designators and 4.7 Procedure stat.)

undersøg ord

2.2.1. DIGITS.

<digit> ::= 0|1|2|3|4|5|6|7|8|9

Digits are used for forming numbers, identifiers, and strings.

2.2.2. LOGICAL VALUES.

<logical value> ::= true|false

The logical values have a fixed obvious meaning.

2.3. DELIMITERS.

<delimiter> ::= <operator>|<separator>|<bracket>|<declarator>|<specificator>

<operator> ::= <arithmetic operator>|<relational operator>|<logical operator>|
<sequential operator>

<arithmetic operator> ::= + | - | × | / | ÷ | ↑

<relational operator> ::= < | ≤ | = | ≥ | > | ≠

<logical operator> ::= ≡ | ⊃ | ∨ | ∧ | ¬

⊃ ¬

a It should be particularly noted that throughout the reference language underlining is used for defining ~~[illegible]~~ ~~are understood to represent~~ independent basic symbols (see, having no relation to the letters of which they ~~may~~ are be composed. Within the present report underlining will be used for no other purpose

02

<sequential operator> ::= go to | if | then | else | for | do[1]

<separator> ::= , | . | $_{10}$ | : | ; | := | ␣ | step | until | while | comment

<bracket> ::= (|) | [|] | ‘ | ’ | begin | end

<declarator> ::= own | Boolean | integer | real | array | switch | procedure

<specificator> ::= string | label | value ~~comment~~

0

Delimiters have a fixed meaning which for the most part is obvious, or else will be given at the appropriate place in the sequel.

Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

15 Side # 7

3 For two the purpose of including text among the symbols of a program the following "comment" convention ~~holds~~ hold: The ~~string~~ sequence 3 00

; comment <any string not containing ; > ;

is ~~syntactically~~ equivalent to a ;

and begin comment <any string > ; begin

end

2.4. IDENTIFIERS.

2.4.1. Syntax.

<identifier> ::= <letter> | <identifier><letter> | <identifier><digit>

2.4.2. Examples.

```
q
Soup
V17a
a34kTMNs
MARILYN
```

2.4.3. Semantics.

Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures. They may be chosen freely (cf. however section 3.2.4. STANDARD FUNCTIONS).

The same identifier cannot be used to denote two different given objects except when these ~~objects~~ have disjoint scopes as defined by the declarations of the program (cf. section 2.~~10~~9. SCOPE and section 5. DECLARATIONS).

2.5. NUMBERS.

2.5.1. Syntax.

<unsigned integer> ::= <digit> | <unsigned integer><digit>

<integer> ::= <unsigned integer> | +<unsigned integer> | -<unsigned integer>

<decimal fraction> ::= .<unsigned integer>

<exponent part> ::= $_{10}$<integer>

<decimal number> ::= <unsigned integer> | <decimal fraction> |
<unsigned integer><decimal fraction>

<unsigned number> ::= <decimal number> | <exponent part> |
<decimal number><exponent part>

<number> ::= <unsigned number> | +<unsigned number> | -<unsigned number>

2.5.2. Examples.

0	-200.084	$-.083_{10}-02$
177	$+07.43_{10}8$	$-_{10}7$
.5384	$9.34_{10}+10$	$_{10}-4$
+0.7300	$2_{10}-4$	$+_{10}+5$

2

[1] do is used in for statements. It has no relation whatsoever to the do of the preliminary report, which is not included in ALGOL 60.

side 6

2.5.3. Semantics.

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

2.5.4. Types.

Integers are of type <u>integer</u>. All other numbers are of type <u>real</u> (cf. section 5.1. TYPE DECLARATIONS).

2.6. STRINGS.

sequence

2.6.1. Syntax.

```
<proper string> ::= <any string of basic symbols not containing ‘ or ’>|
                    <empty>
<open string> ::= <proper string>|‘<open string>’ |<open string><open string>
<string> ::= ‘<open string>’
```

2.6.2. Examples.

```
‘5k,,-‘[[[‘^=/:‘Tt’’
‘..This_is_a_‘string’’
```

sequences

2.6.3. Semantics.

In order to enable the language to handle arbitrary ~~strings~~ of basic symbols the string quotes ‘ and ’ are introduced. The symbol _ denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (cf. 3.2. ~~Function designators~~ and 4.7. Proc-st.)

5

Function designators and S SCOPES

B Bau

2.7. ~~QUANTITIES, KINDS AND TYPES.~~

The following kinds of quantities are distinguished: simple variables, ~~expressions~~, arrays, labels, switches, and procedures.

~~When the word "type" is used it refers to some of the properties of quantities, in particular those properties which can be declared in the language (cf. 5. DECLARATIONS).~~

and Types

1 footnote 2.8. Values[1] and types.

2.8. VALUES.

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

8

~~Values are associated with certain of the syntactic units as follows:~~ (cf. section 3. EOB)

~~The value associated with an expression is the value of that expression.~~

The value ~~associated with~~ of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf. section 3.1.4.1).

Procedure identifiers and switch identifiers are, in general, incomplete expressions and have no values associated with them. The exception is a procedure identifier of a procedure declaration with an empty formal parameter part (cf. section 5.4. PROCEDURE DECLARATIONS), which is a complete expression. if it defines the value of a function designator.

~~The value associated with a particular syntactic unit will in general change during the execution of the program~~

1.

[1] Some members of the conference have doubts about the usefulness of this concept and the way ~~they are~~ it is applied (cf. section 4.7. Proc-staten). (F. L. Bauer and K. Samelson)

Perlis: delete

NB

2.9. RULES AND NAMES.

A rule is a syntactic unit which defines a value. There exist two species of rules in the language: expressions and array identifiers.

Any rule is understood to have a name. This name is the very syntactic construction defining the rule.

Names are in the same way given to the incomplete expressions (procedure identifiers and switch identifiers).

the declaration for ~~the identifier associated with that~~

cf. 5.3.5

6

9

~~possess~~

2.10. SCOPE.

~~The scope of a block is the set of statements comprising the block.~~

?

The scope of a ~~property of a~~ quantity is the set of statements in which that quantity is ~~declared to have that property~~.

2.11 Normal order of Symbols.

valid

? function ~~values~~ designators and

3. EXPRESSIONS.

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational, expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, elementary arithmetic, relational, logical, and sequential, operators~~and other operators called procedures~~. Since the syntactic definition of both variables and ~~procedures may contain~~ s expressions, the definition of expressions, and their constituents, is necessarily recursive.

function designators

NB

```
<expression> ::= <arithmetic expression>|<Boolean expression>|
                 <designational expression>
```

3.1. VARIABLES.

3.1.1. Syntax.

```
<variable identifier> ::= <identifier>
<simple variable> ::= <variable identifier>
<subscript expression> ::= <arithmetic expression>
<subscript list> ::= <subscript expression>|<subscript list>,<subscript expression>
<array identifier> ::= <identifier>
<subscripted variable> ::= <array identifier>[<subscript list>]
<variable> ::= <simple variable>|<subscripted variable>
```

3.1.2. Examples.

```
epsilon
detA
a17
Q[7, 2]
x[sin(n×pi/2), Q[3, n, 4]]
```

the value of

Type Type

3.1.3. Semantics.

4

NB

~~Variables are designations for single numerical or logical quantities.~~ The type of ~~quantity denoted by~~ a particular variable is defined in the declaration for the variable itself (cf. section 5.1. TYPE DECLARATIONS) or for the corresponding array identifier (cf. section 5.2. ARRAY DECLARATIONS).

4

A variable is a designation given to a single numerical or logical value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements.

3.1.4. Subscripts.

3.1.4.1. Subscripted variables designate quantities which are components of multidimensional arrays (cf. section 5.2. ARRAY DECLARATIONS). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable, and is called a subscript. The complete list of subscripts is enclosed in the subscript brackets [ ]. The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (cf. section 3.3. ARITHMETIC EXPRESSIONS).

3.1.4.2. ~~Subscript expressions must be of type integer, and~~ the value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. section 5.2. ARRAY DECLARATIONS). NB

~~3.1.4.3. A subscripted variable is of the same type as the array of which it is a component (cf. section 5.2. ARRAY DECLARATIONS).~~

3.2. ~~PROCEDURE VALUES.~~ Function designators.

3.2.1. Syntax.

<procedure identifier> ::= <identifier>

<actual parameter> ::= <string>|<expression>|<array identifier>|<switch identifier>|<procedure identifier>

<letter string> ::= <letter>|<letter string><letter>

<parameter delimiter> ::= , |)<letter string> :(

<actual parameter list> ::= <actual parameter>|<actual parameter list><parameter delimiter><actual parameter>

<actual parameter part> ::= <empty>|(<actual parameter list>)

~~<procedure value>~~ <function designator> ::= <procedure identifier><actual parameter part>

3.2.2. Examples.

```
sin (a - b)
J(v + s, n)
R
S(s - 5)Temperature:(T)Pressure:(P)
Compile(´:= `)Stack:(Q)
```

00

3.2.3. Semantics. Function designators define

~~Procedure values are~~ single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration (cf. section 5.4. PROCEDURE DECLARATIONS) to fixed sets of actual parameters. The ~~rules for~~ actual parameters are given in section 4.7. PROCEDURE STATEMENTS. ... governing specification of

NB

Not ~~every~~ procedure declarations define the value of a function designator

3.2.4. Standard functions.

Certain identifiers should be reserved for the standard functions of analysis, which ~~in the language~~ will be expressed as procedures. It is recommended that this reserved list should contain:

abs(E)	for the modulus (absolute value) of the value of the expression E
sign(E)	for the sign of the value of E (+1 for E>0, 0 for E=0, -1 for E<0)
sqrt(E)	for the square root of the value of E
sin(E)	for the sine of the value of E
cos(E)	for the cosine of the value of E
arctan(E)	for the principal value of the arctangent of the value of E
ln(E)	for the natural logarithm of the value of E
exp(E)	for the exponential function of the value of E (e^E).

These functions are all understood to operate indifferently on arguments both of type real and integer. They will all yield function values of type real, except for sign(E) which will have values of type integer. In a particular representation these functions may be available without explicit declarations (cf. section 5. DECLARATIONS).

(variables, arrays, procedures, switch)

3.2.5. ~~Transfer procedures~~.

It is understood that ~~transfer procedures~~ between any pair of ~~recognized entities~~ quantities and expressions may be defined. Among the standard ~~procedures~~ functions it is recommended that there be one, namely

entier(E),

which "transfers" an expression of real type to one of integer type, and assigns to it the value which is the largest integer not greater than the value of E. 00

~~The converse transformation from integer to real type is understood to be built into the operations of the language~~ (cf. section 3.2.4. STANDARD FUNCTIONS and section 4.2. ASSIGNMENT STATEMENTS). APs

3.3. ARITHMETIC EXPRESSIONS.

3.3.1. Syntax.

<adding operator> ::= + | -

<multiplying operator> ::= × | / | ÷

<primary> ::= <unsigned number>|<variable>|<~~procedure value~~ function designator>|
(<arithmetic expression>)

<factor> ::= <primary>|<factor>↑<primary>

<term> ::= <factor>|<term><multiplying operator><factor>

<simple arithmetic expression> ::= <term>|<adding operator><term>|
<simple arithmetic expression><adding operator><term>

OK Rut Bau Sam

<if clause> ::= if <Boolean expression> then

~~<arithmetic if expression> ::= <if clause><simple arithmetic expression>|
<if clause>(<arithmetic expression>)~~

<arithmetic expression> ::= <simple arithmetic expression>|
~~<arithmetic if expression>|~~
~~<arithmetic if expression>~~else<arithmetic expression>
<if clause><simple arithmetic expression>

~~3.3.2. Examples.~~

8

A
Alpha
a × sin (omega × t)
$0.57_{10}12$ × a[N × (N - 1)/2, 0]
(A × arctan(y) + Z)↑(7 + Q)
if q then n-1 else n
if a<0 then A/B else if b=0 then B/A else z

type

3.3.3. Semantics.

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for ~~procedure values~~ function designators it is the value arising from the computing rules defining the procedure function (cf. section 5.4. PROCEDURE DECLARATIONS) when applied to the current values longest

as written

of the procedure parameters given in the expression, Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include if clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. section 3.4. BOOLEAN EXPRESSIONS). This selection is made as follows: The Boolean expressions of the if clauses are evaluated one by one in sequence ~~from left to right~~ until one having the value true is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean. The construction:

else ~~<simple arithmetic expression>~~

is equivalent to the construction:

else if true then <simple arithmetic expression>

(~~in case of ambiguity~~ the largest expression found in this position is understood) NB NB

~~If none of the Boolean expressions of the if clauses of an arithmetic expression is true, the value of the arithmetic expression is undefined.~~

3.3.4. Operators and types.

Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types real or integer (cf. section 5.1. TYPE DECLARATIONS). The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules:

3.3.4.1. The operators +, -, and × have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be integer if both of the operands are of integer type, otherwise real.

3.3.4.2. The operations <term>/<factor> and <term>÷<factor> both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. section 3.3.5). Thus for example

a/b × 7/(p - q) × v/s

means

$((((a \times (b^{-1})) \times 7) \times ((p - q)^{-1})) \times v) \times (s^{-1})$

The operator / is defined for all four combinations of types real and integer and will yield results of real type in any case. The operator ÷ is defined only for two operands both of type integer and will yield a result of type integer defined as follows:

a ÷ b = sign(a/b) × entier(abs(a/b))

(cf. sections 3.2.4 and 3.2.5).

3.3.4.3. The operation <factor>↑<primary> denotes exponentiation, where the factor is the base and the primary is the exponent. Thus for example

$2 \uparrow n \uparrow m$ means $(2^n)^m$

while

$2 \uparrow (n \uparrow m)$ means $2^{(n^m)}$

Writing i~~1 and i2~~ for ~~two~~ a numbers of integer type, ~~and~~ r1 ~~and r2~~ for ~~two~~ a numbers of real type the result is given by the following rules:

i1 ↑ i2 For i2>0, i1 × i1 × . . . × i1 (i2 times), of type integer.
For i2=0, 1, of type integer.
For i2<0, if i1=0, undefined.
if i1≠0, 1/(i1 × i1 × . . . × i1) (the denominator has -i2 factors) of type real.

B

Bauer
Sam

and a for a number of either integer or real type

```
i1 ↑ r2    For i1>0, exp(r2 × ln(i1)), of type real.
           For i1=0, if r2>0, 0.0 of type real.
                     if r2≤0, undefined.
           For i1<0, always undefined.
r1 ↑ i2    For i2>0, r1 × r1 × . . . × r1  (i2 times), of type real.
           For i2=0, 1.0, of type real.
           For i2<0, if r1=0.0, undefined
                     if r1≠0.0, 1.0/(r1 × r1 × . . . × r1)  (the denominator
                     has -i2 factors), of type real.
r1 ↑ r2    For r1>0, exp(r2 × ln(r1)), of type real.
           For r1=0, if r2>0, 0.0, of type real.
                     if r2≤0, undefined.
           For r1<0, always undefined.
```

3.3.5. Precedence of operators.

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.3.5.1. According to the syntax given in section 3.3.1 the following rules of precedence hold:

first: ↑
second: × / ÷
third: + -

3.3.5.2. The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.

3.3.6. Arithmetics of <u>real</u> quantities.

Numbers and variables of type <u>real</u> must be interpreted in the sense of numerical analysis, i.e. as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

3.4. BOOLEAN EXPRESSIONS.

3.4.1. Syntax.

```
<relational operator> ::=  < | ≤ | = | ≥ | > | ≠
<relation> ::= <arithmetic expression><relational operator><arithmetic expression>
<Boolean primary> ::= <logical value>|<variable>|~~<procedure value>~~ function designator|
                  <relation>|(<Boolean expression>)
<Boolean secondary> ::= <Boolean primary>| ¬<Boolean primary>
<Boolean factor> ::= <Boolean secondary>|<Boolean factor>∧<Boolean secondary>
<Boolean term> ::= <Boolean factor>|<Boolean term>∨<Boolean factor>
<implication> ::= <Boolean term>|<implication> ⊃ <Boolean term>
<simple Boolean> ::= <implication>|<simple Boolean> ≡ <implication>
~~<Boolean if expression> ::= <if clause><simple Boolean>|~~
                  ~~<if clause>(<Boolean expression>)~~
```

<Boolean expression> ::= <simple Boolean>|~~<Boolean if expression>~~|
~~<Boolean if expression>~~ <if clause><simple Boolean expr> else <Boolean expression>

3.4.2. Examples.

```
x = -2
Y>V ∨ z<q
a+b > -5 ∧ z-d > q↑2
p∧q ∨ x≠y
g ≡ ¬a∧b∧¬c∨d∨e⊃¬f
if k<1 then s>w else h≤c
if if if a then b else c then d else f then g else h<k
```

0

3.4.3. Semantics.

A Boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.4.4. Types.

Variables and ~~procedure values~~ function designators entered as Boolean primaries must be declared Boolean (cf. section 5.1. TYPE DECLARATIONS).

3.4.5. The operators.

Relations take on the value true whenever the corresponding relation is satisfied for the expressions involved, otherwise false.

The meaning of the logical operators ¬ (not), ∧ (and), ∨ (or), ⊃ (implies), and ≡ (equivalent), is given by the following function table.

B

b1	false	false	true	true
b2	false	true	false	true
¬b1	true	true	false	false
b1 ∧ b2	false	false	false	true
b1 ∨ b2	false	true	true	true
b1 ⊃ b2	true	true	false	true
b1 ≡ b2	true	false	false	true

3.4.6. Precedence of operators.

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.4.6.1. According to the syntax given in section 3.4.1 the following rules of precedence hold:

first: arithmetic expressions according to section 3.3.5.
second: < ≤ = ≥ > ≠ → 0
third: → ¬
fourth: ∧
fifth: → ∨ → 0
sixth: ⊃
seventh: ≡

3.4.6.2. The use of parentheses will be interpreted in the sense given in section 3.3.5.2.

designator

designator

3.5. DESIGNATIONAL EXPRESSIONS.

3.5.1. Syntax.

<label> ::= <identifier>|<unsigned integer>

<switch identifier> ::= <identifier>

<switch ~~value~~> ::= <switch identifier>[<subscript expression>]

<simple designational expression> ::= <label>|<switch ~~value~~> | (<designational expr>)

~~<designational if expression> ::= <if clause><simple designational expression>|~~

~~<if clause>(<designational expression>)~~ mishal blue

<designational expression> ::= <simple designational expression>|

~~<designational if expression>|~~

<designational ~~if expression~~> else <designational expression>

<if clause><simple

3.5.2. Examples.

```
17
p9
Choose[n - 1]
Town[if y<0 then N else N+1]
if Ab<c then 17 else q[if w≤0 then 2 else n]
```

3.5.3. Semantics.

A designational expression is a rule for obtaining a label of a statement (cf. section 4. STATEMENTS). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the if clauses will select a simple designational expression. If this is a label the desired result is already found. A switch ~~value~~ designator refers to the corresponding switch declaration (cf. section 5.3. SWITCH DECLARATIONS) and by the actual numerical value of its subscript expression selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch ~~value~~ this evaluation is obviously a recursive process.

The value of a switch design one of the

3.5.4. The subscript expression.

~~A switch value~~ is defined only if the subscript expression is of type integer and assumes positive values 1, 2, 3, ... , n, where n is the number of entries in the switch list.

3.5.5. Unsigned integers as labels.

Unsigned integers used as labels have the property that leading zeroes do not affect their meaning, e.g. 00217 denotes the same label as 217.

4. STATEMENTS.

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken by go to statements, which define their successor explicitly, and shortened by conditional statements, which may cause certain ~~sets~~ statements to be skipped.

In order to make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound statements and blocks the definition of statement must necessarily be recursive. Also since declarations, described in section 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1. COMPOUND STATEMENTS AND BLOCKS.

4.1.1. Syntax.

<unlabelled basic statement> ::= <assignment statement>|<go to statement>|
<dummy statement>|<procedure statement>
<basic statement> ::= <unlabelled basic statement>|
<label> : <~~unlabelled~~ basic statement>
<unconditional statement> ::= <basic statement>|<for statement>|
<compound statement>|<block>
<statement> ::= <unconditional statement>|<conditional statement>
~~<compound end>~~ tail ~~::=~~ <statement> end |<statement> ; <compound ~~end~~ tail>
~~<compound tail> ::= <compound end>|~~
~~<compound end><any string not containing end or ; or else>~~
<block head> ::= begin<declaration>|<block head> ; <declaration>
<unlabelled compound> ::= begin <compound tail>
<unlabelled block> ::= <block head> ; <compound tail>
<compound statement> ::= <unlabelled compound>|<label> : <~~unlabelled compound~~ statement>
<block> ::= <unlabelled block>|<label> : <~~unlabelled~~ block>

Struktur skemaer

4.1.2. Examples.

Unlab~~l~~elled basic statements:

```
    a := p + q
    go to Naples
```

Compound statement:

```
    begin x:=0 ; for y:= 1 step 1 until n do x:= x + A[y] ;
          if x>q then go to STOP ~~else~~ if x>w-2 then go to S ;
  Aw: St: W := x + bob end
```

Block:

```
 Q: begin integer i,k ; real w ;
          for i:= 1 step 1 until m do
          for k:= i+1 step 1 until m do
          begin w:= A[i,k] ;
                A[i,k] := A[k,i] ;
                A[k,i] := w end for i and k
    end block Q
```

BÆ

MORFAR
MORMOR
Q ØL
A P JESPER

~~non-local~~

4.1.3. Semantics.

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration (cf. section 5. DECLARATIONS) be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the ~~compound~~. block

0

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be ~~global~~ to it, i.e. will represent the same entity inside the block and in the level immediately outside it. The exception to this rule is presented by labels, which are local to the block in which they occur.

Since a statement of a block may again itself be a block the concepts local and ~~global~~ to a block must be understood recursively. Thus an identifier, which is ~~global~~ to a block A, may or may not be ~~global~~ to the block B in which A is one statement.

~~The string of symbols entered between an end and the following end, ; or else is of no meaning to the program. It may be used to enter arbitrary text.~~

4.2. ASSIGNMENT STATEMENTS.

4.2.1. Syntax.

```
<left part> ::= <variable>/ :=
<left part list> ::= <left part>|<left part list><left part>
<assignment statement> ::= <left part list><arithmetic expression>|
                           <left part list><Boolean expression>
```

vok

4.2.2. Examples.

```
    s := p[0] := n := n + 1 + s
    n := n + 1
    A := B/C - v - q × S
    s[v, k+2] := 3 - arctan(s × zeta)
    V :=  Q > Y ∧ Z
```

4.2.3. Semantics.

Assignment statements serve for assigning the value of an expression to one or several variables. The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4. Types.

All variables of a left part list must be of the same declared type. If the variables are Boolean the expression must likewise be Boolean. If the variables are of type real or integer the expression must be arithmetic. If the type of the arithmetic expression differs from that of the variables, appropriate transfer ~~procedures are~~ understood to be automatically invoked. For transfer from real to integer type the transfer procedure is understood to yield a result equivalent to

    entier(E + 0.5)

where E is the value of the expression.

fun

form

4.3. GO TO STATEMENTS.

4.3.1. Syntax.
<go to statement> ::= go to <designational expression>

4.3.2. Examples.
go to 8
go to exit[n + 1]
go to Town[if y<0 then N else N+1]
go to if Ab<c then 17 else q[if w<0 then 2 else n]

4.3.3. Semantics.
A go to statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus the next statement to be executed will be the one having this value as its label.

4.3.4. Restriction.
Since labels are inherently local, no go to statement can lead from outside into a block.

4.3.5. Go to an undefined~~.~~ Switch ~~value~~ designator
A go to statement is equivalent to a dummy statement designator if ~~the value of~~ the designational expression is an undefined. Switch ~~value~~.
(This also gives a meaning to a go to a label which does not exist. Do we really want this - PN)
~~A go to statement to an un~~

4.4. DUMMY STATEMENTS.

Ent
empty st

4.4.1. Syntax.
<dummy statement> ::= <empty>

4.4.2. Examples.
L:
begin . . . . ; John: end

4.4.3. Semantics.
A dummy statement executes no operation. It may serve to place a label.

4.5. CONDITIONAL STATEMENTS.

4.5.1. Syntax.
<if clause> ::= if <Boolean expression> then
<unconditional statement> ::= <basic statement>|<for statement>|
<compound statement>|<block>
<if statement> ::= <if clause><unconditional statement>|<label>:<if ~~clause><uncon~~ statement>
<conditional statement> ::= <if statement>|
<if statement> else <statement>

NB

4.5.2. Examples.

NB

```
    if x>0 then n := n+1 else
    if v>u then V: q:=n+m go to R
    if s<0∨P≤Q then AA: begin if q<v then a:=v/s else y:=2×a end
                 else if v>s then a:=v-q else if v>s+1 then go to S
```

4.5.3. Semantics.

?

Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean expressions.

4.5.3.1. If statement.
The unconditional statement of an if statement will be executed if the Boolean expression of the if clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

4.5.3.2. Effect of else.

Se Rutishauser

In the general case the unconditional statement of the if statement is followed by the delimiter else. This defines the successor of the unconditional statement to be the statement following the statement immediately following else. (It should be noted that this explanation makes essential use of the strict syntactic definition of statement. ~~Thus in particular no statement beginning immediately after an else may also be terminated by the delimiter else.)~~

~~4.5.4. Go to statement as unconditional.
It follows from the above rule that if the unconditional statement preceding an else is a go to statement, else has no effect. It may therefore be replaced by the usual statement delimiter ;~~

else if

4.5.5. Alternative description.

?

The effect of a conditional statement which includes several if clauses may be described alternatively as follows:

The Boolean expressions will be evaluated one after the other in sequence from left to right, until one yielding the value true is found. Then the unconditional statement following this Boolean is executed and the else acts as described above. If no Boolean having the value true is found, the unconditional statement following the last else in the complete conditional statement (if such a construction is included) will be executed. Thus, unless there occurs a go to statement leading from one of the unconditional statements to a label within the conditional statement itself, at most one of the unconditional statements will be executed.

4.5.~~6~~4. Go to into a conditional statement.

The effect of a go to statement leading into a conditional statement follows directly from the above ~~rules.~~ explanation of the effect of else

Conditional statement.

The syntax allows two different forms:

if B_1 then S_1 else if B_2 then S_2 else S_3 ; S4

if B_1 then S_1 else if B_2 then S_2 else if B_3 then S_3 ; S4

4.6. FOR STATEMENTS.

4.6.1. Syntax.

```
<for list element> ::= <arithmetic expression>|
     <arithmetic expression> step <arithmetic expression> until
          <arithmetic expression>|
     <arithmetic expression> while <Boolean expression>
<for list> ::= <for list element>|<for list> , <for list element>
<for clause> ::= for <variable> := <for list> do
<for statement> ::= <for clause><statement>|
     <label> : <for clause><statement>
```

4.6.2. Examples.

```
     for q := 1 step s until n do A[q] := B[q]
     for k := 1, V1 × 2 while V1 < N do
          for j := A + B, L, 1 step 1 until N, C + D do A[k,j] := B[k,j]
```

4.6.3. Semantics.

A for clause causes the statement S which it precedes to be repeatedly executed zero or more times ~~while both the following conditions remain true:~~ Paris

~~1) The for clause is not exhausted, i.e., further values remain to be assigned to its controlled variable.~~

~~2) No exit out of S has occurred.~~

In addition to causing S to be repeated, the for clause assigns a sequence of values to its controlled variable as described in sec tion 4.6.4 below.

4.6.4. The for list elements.

The for list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the for list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of for list elements and the corresponding execution of the statement S are given by the following rules:

4.6.4.1. Arithmetic expression. This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

4.6.4.2. Step-until-element. An element for the form A step B until C, where A, B, and C, are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional ALGOL statements as follows:

```
        V:=A;
     L1:if (V+B - C)× sign(B) > 0 then begin V:=V-B go to Element exhausted;
        Statement S;
        V:= V + B;      V+V+B
        go to L1;
```

NB

where V is the controlled variable of the for clause and Element exhausted points to the evaluation according to the next element in the for list, or if the step-until-element is the last of the list, to the next statement in the program.

4.6.4.3. While-element. The execution governed by a for list element of the form E *while* F, where E is an arithmetic and F a Boolean expression, is most concisely described in terms of additional ALGOL statements as follows:

```
  L3:V:=E;
     if ¬F then go to Element exhausted;
     Statement S;
     go to L3;
```

where the notation is the same as in 4.6.4.2 above.

4.6.5. The value of the controlled variable upon exit.

Upon exit out of the statement S (supposed to be compound) through a go to statement the value of the controlled variable will ~~remain~~ be the same as it was ~~during~~ immediately preceding the ~~particular~~ execution ~~when~~ of the go to statement. ~~was encountered.~~

If the exit is due to exhaustion of the for list, on the other hand, the value of the controlled variable is undefined after the exit.

4.6.6. Go to leading into a for statement.

The effect of a go to statement, outside a for statement, which refers to a label within the for statement, is ~~undefined~~.

Perlis

4.7. PROCEDURE STATEMENTS.

4.7.1. Syntax.

```
<actual parameter> ::= <string>|<expression>|<array identifier>|
          <switch identifier>|<procedure identifier>
<letter string> ::= <letter>|<letter string><letter>
<parameter delimiter> ::= ,  |)<letter string>:(
<actual parameter list> ::= <actual parameter>|
      <actual parameter list><parameter delimiter><actual parameter>
<actual parameter part> ::= <empty>|(<actual parameter list>)
<procedure statement> ::= <procedure identifier><actual parameter part>
```

4.7.2. Examples.

```
      Spur (A)Order:(7)Result to:(V)
      Transpose (W, v+1)
      Absmax (A, N, M, Yy, I, K)
      Innerproduct(A[t,P,u], B[P], 10, P, Y)
```

These examples correspond to examples given in section 5.4.2.

4.7.3. Semantics.

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4. PROCEDURE DECLARATIONS). Where the procedure body is a statement written in ~~the language~~ ALGOL the effect of this execution will be equivalent to the effect of performing the following operations on the program:

4.7.3.1. Value ~~replacement~~ assignment (call by value).

All formal ~~identifiers~~ parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. VALUES) of the corresponding actual parameters, these assignments being considered as being performed explicitly ~~through suitable statements added at the beginning of~~ before entering the procedure body. These formal parameters will subsequently be treated as local to the procedure body.

NB Rutish.

parameter

or of an identif.

4.7.3.2. Name replacement (call by name).

Any formal ~~identifier~~ not quoted in the value list is ~~understood to be supplied in the form of a rule (cf. section 2.9. RULES AND NAMES).~~ ~~These formal identifiers~~ are replaced, throughout the procedure body, by ~~the names of~~ the corresponding actual parameters, after enclosing these latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

the effect is as though

4.7.3.3. Body replacement and execution.

Finally the procedure body, modified as above, is ~~inserted~~ in place of the procedure statement and executed.

parameters

4.7.4. Actual-formal correspondence.

The correspondence between the actual parameters of the procedure statement and the formal ~~identifiers~~ of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

ALGOL

4.7.5. Restrictions.

For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct statement. ~~in the language.~~

This poses the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1. Strings cannot occur as actual parameters in procedure statements calling procedure declarations having ALGOL 60 statements as their bodies (cf. section 4.7.8).

4.7.5.2. A formal ~~identifier~~ which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

4.7.5.3. ~~A formal~~ identifier which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. Perlis

If the formal identifier is called by value the local array created during the call will in addition have the same subscript bounds as the actual array.

passes

4.7.5.4. A formal ~~identifier~~ which is called by value cannot in general correspond to a switch identifier or a procedure identifier, because these latter do not ~~have~~ values ~~associated with them (cf. section 2.8. VALUES)~~. (the exception is :

4.7.5.5. ~~Type restrictions exist as follows:~~ Any procedure body will be written on the assumption that the types of all formal ~~identifiers~~ are known (they may, or may not, be given through specifications in the procedure heading). Any actual parameter inserted in the procedure statement must be compatible with the type of the corresponding formal ~~identifier~~. ?

if the type is given by specifications

B

5.4.5

heading of the proc. declaration

parameter

or of an identif.

4.7.3.2. Name replacement (call by name).

Any formal ~~identifier~~ not quoted in the value list is ~~understood to be supplied in the form of a rule (cf. section 2.9. RULES AND NAMES). These formal identifiers are~~ replaced, throughout the procedure body, by ~~the names of~~ the corresponding actual parameters, after enclosing these latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

the effect is as though

4.7.3.3. Body replacement and execution.

Finally the procedure body, modified as above, is ~~inserted~~ in place of the procedure statement and executed.

parameters

4.7.4. Actual-formal correspondence.

The correspondence between the actual parameters of the procedure statement and the formal ~~identifiers~~ of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

ALGOL

4.7.5. Restrictions.

For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct statement. ~~in the language~~.

This poses the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1. Strings cannot occur as actual parameters in procedure statements calling procedure declarations having ALGOL 60 statements as their bodies (cf. section 4.7.8).

4.7.5.2. A formal ~~identifier~~ which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

4.7.5.3. ~~A formal~~ identifier which is used within the procedure body as an array identifier can only cor
which is an array identifier of an
If the formal identifier is called
during the call will in addition ha
the actual array.

4.7.5.4. A formal ~~identifier~~ which
ral correspond to a switch identifi
because these latter do not ~~have~~ va
~~section 2.8. VALUES)~~. (the exception

4.7.5.5. ~~Type restrictions exist as~~

If the formal parameter is an array identifier and called for by value, the dimension of the actual parameter must be identical to that of the formal parameter and the local array created during the call will have the same.....actual array.

(s. 21

be written on the assumption that the types of all formal ~~identifiers~~ are known (they may, or may not, be given through specifications in the procedure heading). Any actual parameter inserted in the procedure statement must be compatible with the type of the corresponding formal ~~identifier~~. (if the type is given by specifications)

NB

5.4.5

heading of the proc. declaration

Non local

4.7.6. ~~Global~~ identifiers of the body.

A procedure statement written outside the scope of the procedure itself or of any ~~global~~ non local identifier of the procedure body is undefined.

4.7.7. Parameter delimiters.

All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number being the same. Thus the information conveyed by using the elaborate ones is entirely optional.

4.7.8. Procedure body expressed not in code. Algol

The restrictions imposed on a procedure statement calling a procedure having its body expressed in code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

(note however that the identifier may again be declared

5. DECLARATIONS.

Declarations serve to define certain properties of the identifiers of the program. A declaration for an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes.

Dynamically this implies the following: at the time of a ~~dynamical~~ entry into a block (through the begin, since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through end, or by a go to statement) all identifiers which are declared for the block lose their significance again.

A declaration may be marked with the additional declarator own. This has the following effect: upon a ~~second~~ re entry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined. All identifiers of a program, with the exception of labels and the possible exception of those for standard functions and transfer procedures (cf. sections 3.2.4 and 3.2.5) must be declared. No identifier may be declared more than once in any one block head.

Syntax.

<declaration> ::= <type declaration>|<array declaration>|<switch declaration>|<procedure declaration> ~~| ;<declaration>~~

NB

Non local

4.7.6. Positional relation of procedure statement to the corresponding procedure body.

For each non-local identifier of the body of a procdüre P, there must be a block in whose heading it is declared. All procedure statements ınvolving P must be positionally within all of these blocks.

S22 - non local ... the decl or of

A
identif

4.7.7.
Al
corresp e
statemen nd
their nu g
the elab

4.7.8. Procedure body expressed in code.

The restrictions imposed on a procedure statement calling a procedure having its body expressed in code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

(note however that the identifier may again be declared

5. DECLARATIONS.

Declarations serve to define certain properties of the identifiers of the program. A declaration for an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes.

Dynamically this implies the following: at the time of a n ~~dynamical~~ entry into a block (through the begin, since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through end, or by a go to statement) all identifiers which are declared for the block lose their significance again.

A declaration may be marked with the additional declarator own. This has the following effect: upon a (re ~~second~~ entry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined. All identifiers of a program, with the possible exception of those for standard functions and transfer procedures (cf. sections 3.2.4 and 3.2.5) must be declared. No identifier may be declared more than once in any one block head.

Syntax.

<declaration> ::= <type declaration>|<array declaration>|<switch declaration>|<procedure declaration> | ;<declaration>

NB

the exception of labels and

5.1. TYPE DECLARATIONS.

5.1.1. Syntax.

```
<type list> ::= <simple variable>|<simple variable>,<type list>
<type> ::= real|integer|Boolean
<local or own type> ::= <type>| own <type>
<type declaration> ::= <local or own type><type list>
```

5.1.2. Examples.

```
integer p, q, s
own Boolean Acryl, n
```

~~simple variables~~ values

identifiers

5.1.3. Semantics.

Type declarations serve to declare certain ~~simple variables~~ to represent ~~quantities~~ of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the values **true** and **false**.

[~~In arithmetic expressions integer declared variables will form a subset of real declared ones.~~] ok

For the semantics of own see the fourth paragraph of 5 above

5.2. ARRAY DECLARATIONS.

5.2.1. Syntax.

```
<lower bound> ::= <arithmetic expression>
<upper bound> ::= <arithmetic expression>
<bound pair> ::= <lower bound> : <upper bound>
<bound pair list> ::= <bound pair>|<bound pair list> , <bound pair>
<array segment> ::= <array identifier>[<bound pair list>]|
      <array identifier> , <array segment>
<array list> ::= <array segment>|<array list> , <array segment>
<array declaration> ::= array <array list>|
          <local or own type> array <array list>
```

5.2.2. Examples.

```
array a, b, c[7:n,2:m], s [-2:10]
own integer array A[if c<0 then 2 else 1 : 20]
real array q[-7 : -1]
```

5.2.3. Semantics.

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts and the types of the variables.

5.2.3.1. Subscript bounds.

The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter : The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2. Dimensions.

The dimensions are given as the number of entries in the bound pair lists.

5.2.4. The value of own arrays.

Upon reentry into a block the subscript bounds } OWN

of The identity of subscripted variables which are components of an own array is defined

5.1. TYPE DECLARATIONS.

5.1.1. Syntax.

```
<type list> ::= <simple variable>|<simple variable>,<type list>
<type> ::= real|integer|Boolean
<local or own type> ::= <type>| own <type>
<type declaration> ::= <local or own type><type list>
```

5.1.2. Examples.

```
      integer p, q, s
      own Boolean Acryl, n
```

5.1.3. Semantics.

~~simple variables~~ values identifiers

Type declarations serve to declare certain ~~simple variables~~ to represent quantities of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the values true and false.

~~In arithmetic expres~~ arithmetic expressions any position which can be occupied by a real declared variable may be occuppied by an integer declared variable.

~~of real declared ones.~~ ok

For the semantics of own see the fourth paragraph of 5 above.

5.2. ARRAY DECLARATIONS.

5.2.1. Syntax.

```
<lower bound> ::= <arithmetic expression>
<upper bound> ::= <arithmetic expression>
<bound pair> ::= <lower bound> : <upper bound>
<bound pair list> ::= <bound pair>|<bound pair list> , <bound pair>
<array segment> ::= <array identifier>[<bound pair list>]|
    <array identifier> , <array segment>
<array list> ::= <array segment>|<array list> , <array segment>
<array declaration> ::= array <array list>|
        <local or own type> array <array list>
```

5.2.2. Examples.

```
    array a, b, c[7:n,2:m], s [-2:10]
    own integer array A[if c<0 then 2 else 1 : 20]
    real array q[-7 : -1]
```

5.2.3. Semantics.

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts and the types of the variables.

5.2.3.1. Subscript bounds.

The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter : The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2. Dimensions.

The dimensions are given as the number of entries in the bound pair lists.

5.2.4. The value of own arrays.
Upon reentry into a block the subscript bounds } own
of
The identity of subscripted variables which are components of an own array is defined

non-local

5.2.3.3. Types.
All arrays declared in one declaration are of the same quoted type. If no type declarator is given the type real is understood.

5.2.4. Lower upper bound expressions.

5.2.4.1. The expressions must be of integer type (cf. section 3.3.4).

5.2.4.2. The expressions can only depend on variables and procedures which are ~~global~~ to the block for which the array declaration is valid. Consequently in the outermost block of a program only array declarations with ~~fixed~~ constant bounds may be declared. the values of

NB

5.2.4.3. An array is defined only when all upper subscript bounds are not smaller than √the corresponding lower bounds.

√those of

those of

24

5.2.4.4. The expressions will be evaluated once at each entrance into the block.

5.3. SWITCH DECLARATIONS.

5.3.1. Syntax.

```
<switch list> ::= <designational expression>|
                  <switch list> , <designational expression>
<switch declaration> ::= switch <switch identifier>:=<switch list>
```

5.3.2. Examples.

```
switch S := S1, S2, Q[m], if v>-5 then S3 else S4
switch Q := p1, w
```

5.3.3. Semantics.
A switch declaration defines the values corresponding to a switch identifier. These values are given ~~one by one~~ in the switch list. as the values of the designational expressions entered With each ~~designational expression in the switch list~~ of these [illegible] there is associated a positive integer, 1, 2, ... , obtained by counting the items in the list from left to right. The switch value corresponding to a given value of the subscript expression (cf. section 3.5. DESIGNATIONAL EXPRESSIONS) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4. Evaluation of expressions in the switch list.
An expression in the switch list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

5.3.5. Influence of scopes.
Any reference to a switch value from outside the scope of any quantity entering into the designational expression for this particular value is undefined.

5.4. PROCEDURE DECLARATIONS.

5.4.1. Syntax.

<formal ~~identifier~~ parameter> ::= <identifier>
<formal parameter list> ::= <formal ~~identifier~~ parameter>|
    <formal parameter list><parameter delimiter><formal ~~identifier~~ parameter>
<formal parameter part> ::= <empty>|(<formal parameter list>)
<identifier list> ::= <identifier>|<identifier list> , <identifier>
<value part> ::= value <identifier list> ; |<empty>
<specifier> ::= string|<type>|array|<type> array | label | switch |
    procedure | <type> procedure
<specification part> ::= <empty>|
    <specifier><identifier list> ; |
    <specifier><identifier list> ; <specification part>
<procedure heading> ::= <procedure identifier><formal parameter part>;
    <value part><specification part>
<procedure body> ::= <statement>|<code>
<procedure declaration> ::=
    procedure <procedure heading><procedure body>|
    <type> procedure <procedure heading><procedure body>

? perlis

NB
value
køres samm
mede
specifiers

5.4.2. Examples.

```
procedure Spur(a)Order:(n)Result:(s) ; value n ;
array a ; integer n ; real s ;
begin integer k ;
s := 0 ;
for k := 1 step 1 until n do s := s + a[k,k]
end

procedure Transpose(a)Order:(n) ; value n ;
array a ; integer n ;
begin real w ; integer i, k ;
for i := 1 step 1 until n do
    for k := 1+i step 1 until n do
    begin w := a[i,k] ;
          a[i,k] := a[k,i] ;
          a[k,i] := w
    end
end Transpose
```

~~Boolean procedure Boolfromreal(q) ; real q ;~~
~~Boolfromreal := if q≥0 then true else false~~

q ≥ 0

```
integer procedure Step(u) ; real u ;
Step := if 0≤u∧u≤1 then 1 else 0

procedure Absmax(a)size:(n,m)Result:(y)Subscripts:(i,k) ;
comment The absolute greatest element of the matrix a, of size n by m
is transferred to y, and the subscripts of this element to i and k ;
array a ; integer n, m, i, k ; real y;
begin integer p, q ;
y := 0 ;
for p := 1 step 1 until n do for q := 1 step 1 until m do
if abs(a[p,q]) > y then begin y:=abs(a[p,q]); i:=p; k:=q end end Absmax
```

through proc. st. or function designator

```
procedure Innerproduct(a,b)Order:(k,p)Result:(y) ; value k ;
integer k,p ; real y,a,b ;
begin real s ;
s := 0 ;
for p := 1 step 1 until k do s := s + a × b ;
y := s
end Innerproduct
```

non local segment parameters parameters

5.4.3. Semantics.

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which may be activated from other parts of the ~~program.~~ block in the head of which the proc decl. appears Associated with the body is a heading, which specifies certain identifiers occurring within the body to be formal. Formal ~~identifiers~~ in the procedure body will, whenever the procedure is activated (cf. section 3.2. PROCEDURE VALUES and section 4.7. PROCEDURE STATEMENTS) be assigned the values of or replaced by ~~the names of~~ actual parameters. Identifiers in the procedure body which are not formal will be either local or ~~global~~ non local to the body depending on whether they are declared within the body or not. Those of them which are ~~global~~ to the body may well be local to the block in the head of which the procedure declaration appears.

values of function designators

5.4.4. ~~Procedure values.~~

For a procedure declaration to define ~~a procedure value~~ the value of a function designator there must, within the procedure body, occur an assignment of a value to the procedure identifier. The type of ~~the procedure~~ This value is given by the type declarator inserted as the first symbol in the ~~heading.~~ declar. ~~If no type is given, the declarator real is understood.~~ value

Any other ...

parameters

5.4.5. Specifications.

In the heading a specification part, giving information about the kinds and types of the formal ~~identifiers~~, may be included. This is considered to be an optional feature which may be omitted.

Real understood

If a specification part is included it must be complete except for the specifications of structure

(I would be strongly in favour of making the specification part a regular, and not just an optional, feature. PN)

5.4.6. Code as procedure body.

It is understood that the procedure body may be expressed in non-ALGOL language. Since it is intended that the use of this feature should be entirely a question of hardware representation, no further rules concerning this code language can be given within the reference language.

real <ident list>

which may be omitted

The procedure body, being a part of a declaration, can only be activated through