9. Jan. 1960 1960 Jan 14 5

ALGOL 60.
Draft report by Peter Naur, Regnecentralen, Copenhagen.

Description of the reference language.

NB

1. STRUCTURE OF THE LANGUAGE.

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, selfcontained units of the language - explicit formulae - called assignment statements.

To show the flow of computational processes, certain nonarithmetic statements and statements clauses are added which may describe e.g., alternatives, or 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.

101

Statements may be 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.

Sequences of statements may be combined into compound statements by insertion of statement brackets. The range of validity of a declaration is one statement (simple or compound).

NB Input - output

A program is a selfcontained compound statement, i.e. a compound statement which makes no use of ~~objects which are not defined within itself~~.

In the sequel explicit rules - and associated interpretations - will be given describing the syntax of the language. Any sequence of symbols to which these rules do not assign a specific interpretation will be considered to be undefined. Specific translators may give such sequences different interpretations.

102

NB

1.1. Formalism for syntactic description.

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

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>.

<empty> ::= the null string of ~~symbols~~ character

2. BASIC SYMBOLS. Unsigned integers and identifiers.

The reference language is built up from the following basic symbols:
<basic symbol> ::= <letter>|<digit>|<delimiter>

2.1. Letters.

<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

2.2. Digits.

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

~~2.2. Logical values~~
~~<logical value> ::= true | false~~

2.3. Delimiters.

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

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

<arithmetic operator> ::= + | - | × | / | ÷ | ↑
<relational operator> ::= < | ≤ | = | ≥ | > | ≠
~~<logical operator> ::= ¬ | ∨ | ∧ | ≡ | ⊃~~
<sequential operator> ::= go to | ~~do~~ | ~~return~~ | ~~stop~~ | for | if | ~~if either~~ | ~~or if~~ | yields | else

<separator> ::= , | : | ; | := | ~~=:~~ | ~~→~~ | 10 |

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

<declarator> ::= procedure|array|switch|<type>|comment|function|~~local~~ real|
label|input|output|format|~~global~~

~~Of these symbols, letters do not have individual meaning. Figures and~~ 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.

Strings of letters and figures enclosed by delimiters represent new entities. However only two types of such strings are admissible:

2.4. Unsigned integers.

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

Examples: 8
1230
00217
0

Unsigned integers may represent the positive integers with the conventional meaning, but have other applications as well.

2.5. Identifiers.

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

Examples: q
Soup
V17a
a34kTMNs
MARILYN

Identifiers have no inherent meaning, but serve for identifying purposes only. They may be chosen freely (cf however section 3.3.4 on standard functions). The same identifier may be used to denote different entities, only if they are declared in different compound statements or blocks.

3. EXPRESSIONS.

The processes which the algorithmic language is intended to describe are given primarily by arithmetic and logical expressions. Constituents of these expressions, except for certain delimiters, are numbers, variables, elementary arithmetic operators and relations, and other operators called functions. Since the description of both variables and functions may contain expressions, the definition of expressions, and their constituents, is necessarily recursive.

Expressions fall in the following classes:

<general arithmetic expression> ::=
<arithmetic expression>|<conditional arithmetic expression>
<general Boolean expression> ::=
<Boolean expression>|<conditional Boolean expression>
<general designational expression> ::=
<designational expression>|<conditional designational expression>
<expression> ::=
<general arithmetic expression>|<general Boolean expression>

The following are the units from which expressions are constructed.

3.1. UNSIGNED NUMBERS.

3.1.1. Syntactic definition.

<unsigned integer> ::= <digit>|<unsigned integer><digit>
<decimal fraction> ::= .<unsigned integer>
<exponent part> ::= 10<unsigned integer>
<decimal number> ::= <unsigned integer>|<decimal fraction>|
<unsigned integer><decimal fraction>
<unsigned number> ::= <decimal number>|<exponent part>|
<decimal number><exponent part>

3.1.2. Examples.

0	200.084	$.083_{10}-02$
17	$07.43_{10}8$	10^{7}
.5384	$9.34_{10}+10$	10^{-4}
0.073	$2_{10}-4$	10^{+5}

3.1.3. Semantics.

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

3.2. VARIABLES.

3.2.1. Syntax.

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

3.2.2. Examples.

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

3.2.3. Semantics.

Variables are designations for arbitrary scalar quantities, e.g., numbers as in elementary arithmetic, unless otherwise specified. However, certain declarations (cfr. type declarations, section) may specify them to be of a special type, e.g., integral, or Boolean. Boolean (or logical) variables may assume only the two values true and false.

Identifiers for variables may be chosen freely.

3.2.4. Subscripts.

3.2.4.1. Subscripted variables designate quantities which are components of multidimensional arrays (cfr. array declarations, section). Each general 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 (cfr. arithmetic expression, section 3.4.3).

3.2.4.2. Subscript expressions must be integer valued in the mathematical sense (cfr. section 4.2.3) and must assume values within the subscript bounds of the array (cfr. section).

3.2.4.3. A subscripted variable is of the same type as the array of which it is a component (cfr. section).

3.3. FUNCTION VALUES.

3.3.1. Syntax.

```
<function> ::= <identifier>
<procedure> ::= <identifier>
<input parameter> ::= <array>|<procedure>|<function>|<general arithmetic
        expression>
<input list> ::= <input parameter>|<input list> , <input parameter>
<blank> ::= <the null string of symbols>
<input part> ::= <blank>|(<input list>)
<function value> ::= <function><input part>
```

3.3.2. Examples.

```
sin (a - b)
J(v + s, n)
R
```

3.3.3. Semantics.

Function values are single numbers which result through the application of given sets of rules defined by a function declaration (cfr. section) to fixed sets of parameters. A syntactic definition of parameters is given in the sections on function declarations. If the function is defined by a short function declaration, the parameters employed in any use of the function are arithmetic expressions. Admissible parameters for functions defined by long function declarations are the same as admissible input parameters of procedures as listed in section on procedure statements.

3.3.4. Standard functions.

Identifiers designating functions may be chosen according to taste just as in the case of variables. However, certain identifiers should be reserved for the standard functions of analysis. This reserved list should contain:

These functions are all understood to operate on arguments of type real. They will all yield function values of type real, except for sign(E) which will have values of type integer, -5-

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, -1 for E<0)
entier(E)	for the largest integer not greater than the value of E
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 arcus tangens 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).

In a particular representation these functions may be available without explicit declarations (cf. section 5 Declarations).

3.4. ARITHMETIC EXPRESSIONS.

3.4.1. Syntax.

<factor> ::= <unsigned number>|<function value>|<variable>|(<arithmetic expression>)|<factor>↑<arithmetic expression>

<term> ::= <factor>|<term>×<factor>|<term>/<factor>

<arithmetic expression> ::= <term>|+<term>|-<term>|<arithmetic expression> + <term>|<arithmetic expression> - <term>

3.4.2. Examples.

A

Alpha

a × sin (omega × t)

$0.57_{10}12 \times a[N \times (N - 1)/2, 0]$

(A × arctan(y) + Z)↑(7 + Q)

3.4.3. Operational meaning.

An arithmetic expression is a rule for computing a value (a number) by executing the indicated arithmetic operations on the actual numerical values of the constituents of the expression. This value is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function values it is the value arising from the computing rules defining the function (cfr. function declarations section) when applied to the current values of the function parameters given in the expression. These variables and functions must not be declared Boolean (cfr. section).

3.4.4. The operators.

The operators +, -, ×, / have the conventional meaning with the additional rule that / is interpreted as a multiplication by the reciprocal of the factor of the term, with due regard to the rules of precedence (cfr. section 3.4.5). Thus for example

a/b × 7/(p - q) × v/s

means

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

The parentheses ↑ denote exponentiation, where the leading factor is the base and the expression enclosed in parentheses is the exponent. Thus

2 ↑ 2 ↑ n means $2^{(2^n)}$

(2 ↑ 2) ↑ n means $(2^2)^n$

3.4.5. Precedence of operators.

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

3.4.5.1. Parentheses are evaluated separately.

3.4.5.2. For operators, the conventional rule of precedence applies. holds:
first: ↑
second: × /
third: + -.

The proper interpretation of expressions can always be arranged by appropriate positioning of parentheses.

3.5. BOOLEAN EXPRESSIONS.

3.5.1. Syntax.

<general arithm expr A> ::= <general arithmetic expression>
<relation> ::= <general arithmetic expression><relational operator>
<general arithm expr A>
<Boolean term> ::= 0|1|<relation>|<function value>|<variable>|
(<Boolean expression>)| ¬<Boolean term>
<Boolean expression> ::= <Boolean term>|<Boolean expression>∨<Boolean term>|
<Boolean expression>∧<Boolean term>|
<Boolean expression>≡<Boolean term>

3.5.2. Examples.

x = 0
X>0 ∨ y>0
¬p∧q ∨ x≠y

3.5.3. Operational meaning.

Boolean expressions are analogous to arithmetic expressions. However, the values of the constituents are confined to the truth values true and false represented if so desired, by the integers 1 (true) and 0 (false). Variables and functions except those which appear in relations, must be declared Boolean (cfr. section 5.1. TYPE DECL. ...)

3.5.4. The operators.

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

The meaning of the logical operators ¬ (not), ∨ (or), ∧ (and), and ≡ (equivalent) is given by the following function table, where 0 stands for false and 1 for true:

B1	0	0	1	1
B2	0	1	0	1
¬ B1	1	1	0	0
B1 ∨ B2	0	1	1	1
B1 ∧ B2	0	0	0	1
B1 ≡ B2	1	0	0	1

3.5.5. Precedence of operators.

The rules are analogous to those of section 3.4.5 with section 3.4.5.2 replaced by

3.5.5.1. The following rule of precedence holds:
first: ¬
second: ∧
third: ∨
fourth: ≡

3.6. DESIGNATIONAL EXPRESSIONS.

3.6.1. Syntax.

<label> ::= <identifier>|<unsigned integer>
<switch> ::= ~~<identifier>~~
<switch value> ::= <switch>[<subscript expression>]
<designational expression> ::= <label>|<switch value>

3.6.2. Examples.

17
p9
Choose[n - 1]
~~Town[y<0 yields N else 1 yields N+1]~~

3.6.3. Semantics.

The value of a designational expression is always finally a label of a statement (cfr. section 4 STATEM...). In the case of a label this value is given directly. A switch value refers to the corresponding switch declaration (cfr. section 5. SWITCH) and by the actual (current) 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.

3.6.4. The subscript expression.

~~The~~ subscript expression ~~must assume~~ only positive integral values 1, 2, 3, ..., n, where n is the number of entries in the switch declaration list.

3.6.5. Unsigned integers as labels.

Unsigned integers used as labels ~~are treated as numbers. Thus label 00217 is not different from label 217.~~

3.7. CONDITIONAL EXPRESSIONS.

3.7.1. Syntax.

<conditional arithmetic expression> ::=
    <general Boolean expression> yields <arithmetic expression>|
    <conditional arithmetic expression> else <general Boolean expression>
        yields <arithmetic expression>
<conditional Boolean expression> ::=
    <general Boolean expression> yields <Boolean expression>|
    <conditional Boolean expression> else <general Boolean expression>
    yields <Boolean expression>
<conditional designational expression> ::=
    <general Boolean expression> yields <designational expression>|
    <conditional designational expression> else <general Boolean expression>
    yields <designational expression>

3.7.2. Examples.

x>0 yields -x else 1 yields x

3.7.3. Semantics.

The value of a conditional expression is found as follows: The Boolean expressions preceding the delimiters yields are evaluated in sequence from

left to right until one is found which is true. Then the value of the conditional expression is the expression immediately following the yields following this Boolean expression.

4. STATEMENTS.

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

4.1.1. Syntax.

<unlabled basic statement> ::= <assignment statement>|<go to statement>|
<do statement>|<procedure statement>|<output statement>|<dummy statement>
<not end> ::= <any string not containing the delimiters end and ;>
<compound statement> ::= begin <statement list> end <not end>
<statement> ::= <unlabled statement>|<label> : <unlabled statement>
<unlabled statement> ::= <unlabled basic statement>|<compound statement>
<statement list> ::= <statement>|<statement list> ; <statement>

4.1.2. Examples.

Unlabled basic statements:

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

Compound statement:

```
if a > b ; p := p + 1
begin local AA ; AA := p × r ; output V(AA) ;
if (AA > 10) ; go to Y ; x := x × cos(t) - y × sin(t) ;
y := y × cos(t) + x × sin(t) ; Y: output V(r) end Here ends print cycle
```

4.1.3. Semantics.

Every compound statement (defined by the statement bracket begin end or by an if- or for-clause) automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the compound may through a suitable declaration (cfr. section 5) be specified to be local to the compound in question. This means (a) that the entity represented by this identifier inside the compound has no existence outside it and (b) that any entity represented by this identifier outside the compound is completely inaccessible inside the compound.

Identifiers (except those representing labels) occurring within a compound statement and not being declared local to this compound will be global to this compound, i.e. will represent the same entity inside the compound and in the level immediately outside it. The exception to this rule is

presented by labels, which are local to the compound in which they occur.

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

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

4.2. ASSIGNMENT STATEMENTS.

First description: no simultaneous assignments.

4.2.1. Syntax.

<assignment statement> ::= <variable> := <expression>

4.2.2. Examples.

```
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 a variable. This process must be understood as follows:

Numbers and variables 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 will realize arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analyis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

By means of a type declaration (section) a variable or function may be declared to belong to a certain class. An expressions containing variables or functions of different types must in general be assumed to have the type which mathematically speaking will embrace it in all cases. In special cases, however, the type of the expression may, for mathematical reasons, and in a given context, be assumed to belong to a more restricted type. If, and only if, this is the case may an expression be assigned to a variable of the more restricted type. Depending on the characteristics of a specific hardware representation such an assignment may or may not give rise to special precautions, such as rounding, and this again may, in the particular representation, cause that the assignment statement will yield a meaningful result even if the expression is not of the restricted type. It should be stressed, however, that this must be considered as an entirely incidental circumstance which in no way changes the above strict rule, that only mathematically correct assignments are permitted in the language.

4.2.4. The relations of Types.

The relations of the types integer and Boolean must be understood as follows:

Boolean declared variables form a subset of integer declared ones.

Integer declared variables form a subset of real declared ones.

Alternative description, simultaneous assignments.

4.2.1. Syntax.
<left part> ::= <variable> :=
<left part list> ::= <left part>|<left part list><left part>
<assignment statement> ::= <left part list><expression>

4.2.2. Examples.
S := p[0] := n := n + 1 + S
A := B/C - v-q × s

4.2.3. Semantics. As above with trivial modifications.

4.2.4. Evaluation.
The meaning of the multiple assignments is that the expression is evaluated once and then assigned to all the left part variables.

4.2.5. Types.
All variables of a left part list must be of the same declared type.

4.2.6. The relations of types.
As above.

4.3. GO TO STATEMENT.

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

4.3.2. Examples.
go to 8
go to exit[n + 1]

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 as the value of a general 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 may lead from outside into a compound statement.

4.4. INPUT STATEMENTS.

4.4.1. Syntax.
<input element> ::= <variable>|<array>
<input statement list> ::= <input element>|<input list> , <input element>
<input format> ::= <identifier>
<input statement> ::= input <input format>(<input statement list>)|

4.4.2. Examples.
input Tape (p[n + 1], v)
input V (Q[v, s-2], R[2])

4.4.3. Semantics.
An input statement serves to assign numbers expressed in an exterior medium to variables. The exterior medium will be given by the input format declaration (cfr. section) defining the meaning of the input format.

4.4.4. Input elements.

The variables and arrays listed in the input statement list must correspond tothe list of formal identifiers given in the input format declaration in precisely the same manner as the procedure output parameters of a procedure statement must correspond to the formal identifiers of the procedure declaration (cfr. section 4. cases 7.1, 7.2 8).

4.5. OUTPUT STATEMENTS.

4.5.1. Syntax.

<output element> ::= <expression>|<array>
<output statement list> ::= <output element>|
<output statement list> , <output element>
<output format> ::= <identifier>
<output statement> ::= output <output format>(<output statement list>)

4.5.2. Examples.

output Typewriter (s[1], s[2], v+q)
output drum (Q)

4.5.3. Semantics.

An output statement serves to transfer the values of expressions to an exterior medium. The exterior medium will be given by the output format declaration (cfr. section 5.) defining the meaning of the output format.

4.5.4. Output elements.

The expression and arrays listed in the output statement must correspond to the list of formal identifiers given in the output format declaration in precisely the same manner as the procedure input parameters of a procedure statement must correspond to the formal identifiers of the procedure declaration (cfr. section 4. cases 1.1 1.2, 1.3, 1.4, 1.5, and 2).

4.6. STOP STATEMENTS.

4.6.1. Syntax.

<stop statement> ::= stop

4.6.2. Semantics.

A stop statement defines the operational end of a program. It has no successor.

If, in hardware representations, it is desired to define a successor (to be activated in case a suitable signal is provided) this should be the following statement.

4.7. RETURN STATEMENTS.

4.7.1. Syntax.

<return statement> ::= return

4.7.2. Semantics.

A return statement defines the operational end of a procedure. Its successor is the statement following the procedure statement being executed.

4.7.3. Occurrence.

Return statements may only occur in the compound statements of procedure declarations and long function declarations.

compounds (cfr sections)

4.8. DUMMY STATEMENTS.

4.8.1. Syntax.

<dummy statement> ::= <blank>

empty

4.8.2. Examples.

L:
begin . . . . ; John: end

4.8.3. Semantics.

A dummy statement executes no operation. It only serves to place a label.

may

IF STATEMENT, FOR STATEMENT, ALTERNATIVE STATEMENT, DO STATEMENT.

Description awaits the decision at Paris.

4. . PROCEDURE STATEMENTS.

4. .1. Syntax.

<open or expression> ::= <blank>|<expression>
<partly open expression list> ::=
<open or expression>|<partly open expression list> , <open or expression>
<array parameter> ::= <array>|<array>[<partly open expression list>]
<output parameter> ::= <variable>|<array parameter>|<general designational expression>
<output list> ::= <output parameter>|<output list> , <output parameter>
<output part> ::= <blank>| =: (<output list>)
<procedure statement> ::= <procedure><input part><output part>

flyt til 3.3.1

identifier

NB <procedure call> ::= < " statement>

4. .2. Examples.

B(a/Q + v, b, M[k,]) =: (S[, n], d, 18)
SR(v[, 8], u - 2, t)
Isyp =: (p, epsilon)
COMPILE

4. .3. Semantics.

A procedure statement serves to initiate (call for) the execution of a procedure compound (cfr. procedure declaration section). The execution, however, is effected as though all formal parameters listed in the formal part of the procedure declaration heading were replaced, throughout the procedure compound, by actual parameters derived from the parameters in the corresponding positions in the procedure statement. In addition, the procedure compound will in certain situations be supplemented by assignment statements inserted before it.

as defined in section 4. .4

4. .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 input-part-output-part of the procedure statement must be

actual parameters first

have

~~identical in form with the formal part of the corresponding procedure declaration. In this way~~ there is defined a one-to-one correspondence between ~~the parameters in the input-output lists of the procedure~~ statement and the identifiers in the formal part of the procedure declaration. This one-to-one correspondence together with the ~~input-output~~ specifications given in the procedure ~~heading~~ (section) give complete information concerning the admissibility of parameters employed in any procedure call. Rules covering all admitted cases are given in sections 4. .8 and 4. .9 together with the corresponding semantic rules.

4. .5. Identity of parameters.
If a formal output parameter is identical to a formal input parameter, this identity must be preserved in the call. The corresponding actual parameter as well as any other parameter entered both as input and output parameter in the procedure call must obviously meet the requirements of both input and output parameters.

4. .6. Actual parameters.
All actual parameters must be defined in the ~~compund statement~~ where the procedure statement occurs.

4. .7. Global parameters of the procedure ~~compound~~.
Identifiers which are global to the procedure ~~compound~~ and which have been used there must not have been redefined (by declaration) in the ~~compound~~ where the procedure statement occurs.

4. .8. Input parameters.
Case 1.1.
Formal specification: typeF (f). (Special case: no specification for f).
Actual parameter: a.
Declaration for actual parameter: typeF (a).
Formal identifier will be replaced by: a.
Execution: procedure compound

Case 1.2.
Formal specification: typeF (f). (Special case: no specification for f).
Actual parameter: a.
Declaration for actual parameter: typeA (a). Quantities of typeA must form a subset of the quantities of typeF.
Formal identifier will be replaced by: g (a unique identifier of typeF).
Execution: g := a ; procedure compound. The assignment g:=a must be permitted, cfr. semantics of assignment statements, section .

Case 1.3.
Formal specification: typeF (f). (Special case: no specification for f).
Actual parameter: a[E1, ... , En]. E1, ... , En are expressions defined in the level where the procedure call is written.
Declaration for actual parameter: typeF array (a[E, ... ,E : E, ... , E].
Formal identifier will be replaced by: a[i, ... , k]. (i, ... , k are unique identifiers of type integer).
Execution: i := E1 ; .... ; k := En ; procedure compound. The assignments must be permited ones, cfr. section .

Case .4.
F.

Case 1.4.
Formal specification: typeF (f). (Special case: no specification for f).
Actual parameter: a[E1, ... , En]. E1, ... , En are expressions defined in the level where the procedure call is written.
Declaration for actual parameter: typeA array (a[E, ... ,E : E, ... , E]. Quantities of typeA must form a subset of the quantities of typeF.
Formal identifier will be replaced by: g (a unique identifier of typeF).
Execution: g := a[E1, ... , En] ; procedure compound.

Case 1.5.
Formal specification: typeF (f). (Special case: no specification for f).
Actual parameter: E (an expression in variables defined in the level where the procedure call is written).
Declaration for actual parameter: none.
Formal identifier will be replaced by: g (a unique identifier of typeF).
Execution: g := E ; procedure compound.

Case 2.1.
Formal specification: typeF array (f[d, ... ,d : d, ... ,d]). (Special case: array (f[d, ... ,d : d, ... ,d])).
Actual parameter: a.
Declaration for actual parameter: typeF array (a[E, ... ,E : E, ... , E]. The number of subscripts and their bounds must be identical for f and a.
Formal identifier will be replaced by: a.
Execution: procedure compound.

Case 2.2.
Formal specification: typeF array (f[d, ... ,d : d, ... ,d]). (Special case: array (f[d, ... ,d : d, ... ,d])).
Actual parameter: a[<partly open expression list>]. The number of open subscript positions must be the same as the number of subscript positions for f. These open positions must have the same bounds as the corresponding positions of f.
Declaration for actual parameter: typeF array (a[E, ... ,E : E, ... , E].
Formal identifier will be replaced by: a. Simultaneously the subscript lists following f will be replaced by the subscript list given in the procedure statement for a supplemented in all open positions by the original subscripts specified for f taken in the same order.
Execution: procedure compound.

Case 3.
Formal specification: typeF function (f(d, ... , d)). Special case: function (f(d, ... , d)).
Actual parameter: a.
Declaration for actual parameter: typeF function a(I, ... , I) .... This declaration must agree with the formal specification with respect to the number and nature of all parameters taken in the same order. If a uses global identifiers these must also be global to .
Formal identifier will be replaced by: a.
Execution: procedure compound.

Case 4.
Formal specification: procedure f(d, ... , d) =: (d, ... , d).
Actual parameter: a.
Declaration for actual parameter: procedure a ..... This declaration must agree with the formal specification with respect to the number and nature of all

parameters taken in the same order. If a uses global identifiers these must also be global to the procedure compound.
Formal identifier will be replaced by: a.
Execution: procedure compound.

Case 5.
Formal specification: input f(d, ... , d).
Actual parameter: a.
Declaration for actual parameter: format a(.....). This format declaration must correspond to an input statement of the form given in the formal specification.
Formal identifier will be replaced by: a.
Execution: procedure compound.

Case 6.
Formal specification: output f(d, .. ,d).
Actual parameter: a.
Declaration for actual parameter: format a(.....). This format declaration must correspond to an input statement of the form given in the formal specification.
Formal identifier will be replaced by: a.
Execution: procedure compound.

4. .8. Output parameter. 9 FS

Case 7.1.
Same rules as for case 1.1.

Case 7.2.
Same rules as for case 1.3.

Case 8.1.
Same rules as for case 2.1.

Case 8.2.
Same rules as for case 2.2.

Case 9.
Formal specification: label (f).
Actual parameter: ~~a. or <unsigned integer>~~ designational expression
Declaration for actual parameter: none, but a or the integer must be a label accessible from the level of the procedure statement.
Formal identifier will be replaced by: a.
Execution: procedure compound.

Case 10.
Formal specification: switch (f:=(d, ... d)).
Actual parameter: a.
Declaration for actual parameter: switch a := (D, ... , D). The switch a must have the same number of positions as f. Variables occurring in any of the D's must be global to the procedure.
Formal identifier will be replaced by: a.
Execution: procedure compound.

4. .9. 10 Return statements.
~~Any return statement within the procedure compound will be replaced by a~~
Da kan procedure ikke bruges på indre compounds!!
Interpretation bør ændres
During ~~a~~ the execution of a procedure statement a return statement defines the statement following the procedure statement as its successor.

go to statement referring, by its label, to the statement following the procedure statement, which, if originally unlabeled, is treated as having been assigned a (unique) label during the replacement process.

5. DECLARATIONS.

Declarations serve to define the properties of the identifiers of the program. A declaration for an identifier is valid for one statement (usually compound). Outside this statement the particular identifier may be used for other purposes.

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

At the time of an exit from a compound (through end, or by a go to statement) all identifiers which are local to the compound lose their significance completely, i.e. at a new entrance into the compound the values of local variables are not defined.

All identifiers of a program must be declared.

The syntax of declarations is as follows:

<declaration> ::= <type declaration>|<array declaration>|<switch declaration>|
<function declaration>|<procedure declaration>|<comment declaration>
<declaration list> ::= <blank>|<declaration>|<declaration list>;<declaration>

No identifier may be declared more than once in any one declaration list.

5.1. TYPE DECLARATION.

5.1.1. Syntax.

<type list> ::= <simple variable>|<simple variable>,<type list>
<type> ::= real|integer|Boolean
<type declaration> ::= <type>(<type list>)

5.1.2. Examples.

integer (p, q, s)
Boolean (Acryl, n)

5.1.3. Semantics.

Type declarations serve to declare certain simple variables to represent quantities of a given class. Real declared variables may assume any positive or negative value, including zero. Integer declared variables may assume all positive and negative integral values, including zero. Boolean declared variables may assume the values true and false.

Integer declared variables form a subset of real declared ones.

5.2. ARRAY DECLARATIONS.

5.2.1. Syntax.

<lower bound> ::= <general arithmetic expression>
<upper bound> ::= <general arithmetic expression>
<lower upper bound list> ::= <lower bound> : <upper bound> |
    <lower bound> , <lower upper bound list> , <upper bound>
<array segment> ::= <array> [<lower upper bound list>] |
    <array> , <array segment>
<array list> ::= <array segment> | <array list> , <array segment>
<array declaration> ::= array <array list> | <type> array <array list>

5.2.2. Examples.

array a, b, c[7,2:n,m], s [-2:10]
integer A[c<0 yields 2 else 1 yields 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 this array in the form of a lower upper bound list. Each of these contains two lists of arithmetic expressions separated by the delimiter :. These two lists give the lower respectively upper bounds for all subscripts taken in order from left to right.

5.2.3.2. Dimensions.

The dimensions are given as half the total numer of entries in the lower upper bound lists.

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 integer valued in the mathematical sense (cfr. section 4.2.3).
5.2.4.2. The expressions can only depend on variables and functions which are global to the statement for which the array declaration is valid.
5.2.4.3. An array is defined only when all upper subscript bounds are not smaller than the corresponding lower bounds.
5.2.4.4. The expressions will be evaluated at each entrance into the statement.

(171) Switch

PROCEDURE DECLARATIONS.

Syntax. Elements of the complete description.

<procedure declaration> ::= procedure <procedure><formal part>
<formal specification part><global specification part>
<procedure compound>

<formal part> ::= <formal input part><formal output part>

<formal specification part> ::= <specification list>|<specification list> ;
<formal specification part>

<specification list> ::= <specification>|<specification><specification list>

<specification> ::= <type specification>|<array specification>|
<label specification>|<switch specification>|<function specification>|
<procedure specification>|<input specification>|<output specification>

<global specification part> ::= <blank>|global <formal specification part>