Kursus ca Jan 25-30 1960

DESCRIPTION OF THE REFERENCE LANGUAGE.

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~~ statement 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. Sequences of statements may be combined into compound statements by insertion of statement brackets.

Statements ~~may be~~ 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. The range of validity of a declaration is one statement (simple or compound).

A list of declarations and the list of statements governed by the declarations given in this particular list are together called a block. NB

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

In the sequel explicit rules - and associated interpretations - will be given describing the syntax and semantics of the language. Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the execution of the computation.

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 order to facilitate the study some formulae have been ~~repeated~~ given in more than one place

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>.

Definition:

<empty> ::= the null string of symbols.

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.

<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).

Letters do not have individual meaning. They are used for forming identifiers and strings.

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>

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

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

0 / NB — <logical operator> ::= ≡ | ⊃ | ∨ | ∧ | ¬

<sequential operator> ::= go to|if|then|else|for|do (1)

1 <separator> ::= , | . | ₁₀ | : | ; | := | ␣ | step | until | while | ~~name~~ NB

<specificator> ::= string | label | value

<bracket> ::= (|) | [|] | ‘ | ’ | begin | end
<declarator> ::= own | Boolean | integer | real | array | switch | procedure

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.

For the purpose of including text among the symbols of a program the following "comment" convention holds: The string

NB comment <any string not containing ; > ;

is syntactically equivalent to a ; NB

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, ~~functions,~~ 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 ~~objects~~ entities except when these ~~objects~~ have disjoint scopes as defined by the declarations of the program (cf. section 2.10. 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	$2_{10}-4$	$+_{10}+5$

2.5.3. 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 integer. All other numbers are of type real (cf. section 5.2. TYPE DECLARATIONS).

sequence
chain
of basic symbols
‘<open string>’

2.6. STRINGS.

2.6.1. Syntax.

NB

<proper string> ::= <any string not containing ‘ or ’>|<empty>
<open string> ::= <proper string>|~~<string>~~|<open string><open string>
<string> ::= ‘<open string>’

2.6.2. Examples.

‘5k,,-‘[[[’∧=/:‘Tt’’
‘..‘This␣is␣a␣string’

2.6.3. Semantics.

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

Entities entities

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

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

NB 175

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).

250

2.8. NAMES.

The name of an identifier is that identifier[1].

NB 0 0

The name of a simple variable, expression[1], array, or procedure is the identifier associated with that simple variable, expression[1], array, or procedure, respectively.

2.9. VALUES.

Values are numbers, logical values, or labels.

251 on The value associated with ~~a variable or~~ an expression is the value of that variable or expression, respectively.

The value associated with an array ~~or a switch~~ is the ordered set of values of ~~its components~~. (identifier) the corresponding array.

The value associated with a procedure is as defined in section PROCEDURE DECLARATIONS.

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.

119

relational ~~and~~ logical, and sequential

3. EXPRESSIONS.

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Bolean, and designational, expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, elementary arithmetic operators ~~and relations~~, nd other operators called functions. Since the description of both variables functions may contain expressions, the definition of expressions, and their ituents, is ~~necessarily recursive~~.

```
<expression> ::= <arithmetic expression>|<Boolean expre~~ssion>|~~
                 <designational expression>
```

3.1. VARIABLES.

.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(nxpi/2), Q[3, n, 4]]
```

3.1.3. Semantics.

single numerical or logical quantities. The particular type of quantity denoted by a particular variable is defined in the declaration for the variable itself (cf. Sec. Types) or for the corresponding array identifier.

Variables are designations for ~~arbitrary scalar quantities, e.g.,~~ numbers as in elementary arithmetic, unless otherwise specified. However, certain declarations (cf. section TYPE DECLARATION) 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.

3.1.4. Subscripts.

3.1.4.1. Subscripted variables designate quantities which are components of multidimensional arrays (cf. section 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 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 ARRAY DECLARATIONS).

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

3.2. PROCEDURE VALUES.

3.2.1. Syntax.

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

3.2.3. Semantics.

Procedure values are single values, which result through the application of given sets of rules defined by a procedure declaration (cf. section PROCEDURE DECLARATIONS) to fixed sets of actual parameters. The rules for actual parameters are given in section

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 <u>real</u> and <u>integer</u>. They will all yield function values of type <u>real</u>, except for sign(E) which will have values of type <u>integer</u>. In a particular representation these functions may be available without explicit declarations (cf. section DECLARATIONS).

3.2.5. Transfer procedures.

It is understood that transfer procedures between any pair of recognized entities may be defined. Among the standard procedures 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.

The converse transformation from <u>integer</u> to <u>real</u> type is understood to be built into the operations of the language (cf. section 3.2.4. STANDARD FUNCTIONS and section 4.1. ASSIGNMENT STATEMENTS).

3.3. ARITHMETIC EXPRESSIONS.

3.3.1. Syntax.

<adding operator> ::= + | -
<multiplying operator> ::= × | / | ÷
<primary> ::= <unsigned number>|<variable>|<procedure value>| (
 (<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>
<if clause> ::= <u>if</u> <Boolean expression> <u>then</u>
<arithmetic if expression> ::= <if clause><simple arithmetic expression>|
 <if clause>(<arithmetic expression>)
<arithmetic expression> ::= <simple arithmetic expression>|
 <arithmetic if expression>|
 <arithmetic if expression><u>else</u><arithmetic expression>

3.3.2. Examples.

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

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 it is the value arising from the computing rules defining the procedure (cf. section PROCEDURE DECLARATIONS) when applied to the current values 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 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 <u>true</u> is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean. The construction:

<u>else</u> <arithmetic expression>

is equivalent to the construction:

<u>else</u> <u>if</u> <u>true</u> <arithmetic expression>.

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
/be simple arithmetic expressions must/of types <u>real</u> or <u>integer</u> (cf. section TYPE DECLARATIONS). The meaning of the basic operators and the types of the expressions to which they lead ~~may be determined from the~~ are given by following rules:

~~3.3.4.1. The operators +, -, and × have the conventional meaning (addition, sub-~~
traction, and multiplication). The type of the expression will be <u>integer</u> if both of the operands are of <u>integer</u> type, otherwise <u>real</u>.

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).

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

$$a/b \times 7/(p - q) \times v/s$$

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

$$a \div b \;= \mathrm{sign}(a/b) \times \mathrm{entier}(\mathrm{abs}(a/b))$$

(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 exampde

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

while

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

Writing i1 and i2 for two numbers of <u>integer</u> type and r1 and r2 for two numbers of <u>real</u> type the result is given by the following rules:

i1 ↑ i2	For i2>0,	i1 × i1 × . . . × i1 (i2 times), of type <u>integer</u>.
	For i2=0,	1, of type <u>integer</u>.
	For i2<0,	if i1=0, undefined.
		if i1≠0, 1/(i1 × i1 × . . . × i1) (THE denominator has -i2 factors) of type <u>real</u>.
12 i1 ↑ r2	For i1>0,	exp(r2 × ln(i1)), of type <u>real</u>.
	For i1=0,	if r2>0, 0.0 of type <u>real</u>.
		if r2≤0, undefined.
	For i1<0,	always undefined.
r1 ↑ i2	For i2>0,	r1 × r1 × . . . × r1 (i2 times), of type <u>real</u>.
	For i2=0,	1.0, of type <u>real</u>.
	For i2<0,	if r1=0.0 undefined
		if r1≠0.0, 1.0/(r1 × r1 × . . . × r1) (the denominator has -i2 factors), of type <u>real</u>.
r1 ↑ r2	For r1>0,	exp(r2 × ln(r1)), of type <u>real</u>.
	For r1=0,	if r2>0, 0.0, of type <u>real</u>.
		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 real quantities.

Numbers and variables of type real 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.

arithm expr

13

3.4. BOOLEAN EXPRESSIONS.

3.4.1. Syntax. <rel. op>

```
<relation> ::= <primary><relational operator><primary>
<Boolean primary> ::= <logical value>|<variable>|<procedure value>|
                      <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> else <Boolean expression>
```

4 B

3.4.2. Examples.

```
x = 0 -2          a+b > -5 ∧ z-d > q↑2
Y>V ∨ z<q
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
```

> NB

NB

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 entered as Boolean primaries must be declared Boolean (cf. section 5.1 TYPE DECLARATIONS).

3.4.5. The operators.

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

4

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

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

/NB

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: Evt aritmetik

first:	< ≤ = ≥ > ≠
second:	¬
third:	∧
fourth:	∨
fifth:	/
sixth:	≡

/NB

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

14

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 if expression> ::= <if clause><simple designational expression>|~~
~~                                  <if clause>(<designational expression>)~~
~~<designational expression> ::= <simple designational expression>|~~
~~                          <designational if expression>|~~
~~                       <designational if expression> else <designational expression>~~
```

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 refers to the corresponding switch declaration (cf. section 5. 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.

tion udgivet

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 declaration 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.

15 s

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 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 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.

unlabelled

4.1.1. Syntax.
<unlabelled basic statement> ::= <assignment statement>|<go to statement>|
<dummy statement>|<procedure statement>
<basic statement> ::= <unlabelled basic statement>|
<label> : <basic statement>
<unconditional statement> ::= <basic statement>|<for statement>|
<compound statement>|<block>
<statement> ::= <unconditional statement>|<conditional statement>
<compound end> ::= <statement> end |<statement> ; <compound end>
<compound tail> ::= <compound end>|
<letter
<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>
<block> ::= <unlabelled block>|<label> : <unlabelled block>

belled

;

4.1.2. Examples.
Unlabled basic statements:
a := p + q
go to Naples

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.

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. / m

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>
```

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
```

§ 3

4.2.3. Semantics.

Assignment statements serve for assigning the value of an expression to one or several variables~~, with the understanding that in a particular execution the expression is evaluated once and the resulting value then assigned to the variables.~~ 251

251

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.

NB

NB Evaluation

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. NB

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

NB 4.3.5. Go to an undefined.
A go to statement is equivalent to a dummy ~~an empty~~ statement if the value of the designational expression is undefined.

NB

(This ~~would~~ also gives a meaning to a go to an undefined label. Do we really want this? PN)

4.4. DUMMY STATEMENTS.

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>
<conditional statement> ::= <if statement>|
        <if statement> else <statement>

4.5.2. Examples.

væk

if x>0 then n := n+1
if v>u then V: q:=n+m else ~~if s>0~~ 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.
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 ;

unconditional (w/ [illegible])

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 statement following the last else in the complete conditional statement will be executed. Thus, unless there occurs a go to statement leading from one of the unconditional statement to a label within the conditional statement itself, only one of the unconditional statements will be executed. at most

4.5.6. Go to into a conditional.

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

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]
```

17

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:

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, th/ for clause assigns a sequence of values to its controlled variable as described in section 4.6.4 below. /e

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

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 the same as it was during the particular execution when 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 ~~statement~~ label within the for statement, is undefined.

4.7. PROCEDURE STATEMENTS.

Beskrivelsen findes senere i rapporten.

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 <u>begin</u>, 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 <u>end</u>, 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 <u>own</u>. This has the following effect: upon a 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 TRANS~~ 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> ~~[illegible]~~

~~;:V: TYPE~~

5.1. TYPE DECLARATIONS.

<local or own type>

5.1.1. Syntax.

<type list> ::= <simple variable>|<simple variable>,<type list>

<~~local~~ type> ::= <u>real</u>|<u>integer</u>|<u>Boolean</u>

<type> ::= <local type>| <u>own</u> <local type>

<type declaration> ::= <~~type~~><type list>

<local or own type>

5.1.2. Examples.

<u>integer</u> p, q, s

<u>own</u> <u>Boolean</u> Acryl, n

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 asume the values <u>true</u> and <u>false</u>.

In arithmetic expressions integer declared variables will form a subset of real declared ones.

95