File No. S360-26
Form C28-6615-0

OS

IBM Systems Reference Library

IBM System/360 Operating System

ALGOL Language

This publication provides the programmer with the information needed to use the IBM System/360 Operating System ALGOL compiler for the solution of scientific and technical problems. ALGOL has been introduced in a number of universities and technical institutes for communication and education purposes. To assist that particular area, the OS/360 ALGOL Compiler is intended to provide a bridge to System /360 for existing ALGOL users. A basic knowledge of the ALGOL language is assumed.

This publication consists of two main parts. The first (section 1 to 5)describes the elements of the ALGOL language, the second (section 6) describes the input/output procedures to be called when using ALGOL.

PREFACE

This publication is based on:

1. the "Revised Report on the Algorithmic language ALGOL 60", published originally in the Communications of the Association for Computing Machinery, volume 6 (1963), page 1, in the Computer Journal, volume 5, number 4 (1963), page 349, and in the Numerische Mathematik, volume 4 (1963), page 420, (some comments in the paper "A list of the remaining trouble spots in ALGOL 60" by D. E. Knuth, published in the ALGOL Bulletin 19 (1965), page 29, were taken into consideration).

2. the "Report on Input-Output Procedures for ALGOL 60", published originally in the Communications of the Association for Computing Machinery, volume 7 (1964), page 628; in the ALGOL Bulletin, number 16, page 9, and in the Numerische Mathematik, volume 6 (1964), page 459.

A form for readers' comments appears at the back of this publication. It may be mailed directly to IBM. Address any additional comments concerning this publication to Programming Systems Publications, Department D58, PO Box 390, Poughkeepsie, NY 12602

CONTENTS

INTRODUCTION

This publication describes the international algorithmic language ALGOL 60 as it is used to write programs to be executed with the System/360 Operating System. ALGOL 60 is a higher level language suitable for expressing a large class of numeric processes in a form sufficiently concise for automatic translation.

Programs written in ALGOL are translated into System/360 machine language code by the System/360 Operating System ALGOL compiler. The compiler analyzes the ALGOL source program and generates an object program that is suitable for a linkage editor processing and subsequent execution. In addition it writes appropriate messages when errors are detected in the source program.

This publication contains a complete description of the language accepted by the compiler. This language is the hardware representation of a proper subset of the full ALGOL language, which is specified in the "Revised Report on the Algorithmic Language ALGOL 60 "[1]. This subset fully contains the ECMA Subset of ALGOL 60 [4], and the SUBSET ALGOL 60 of IFIP [5]. In addition, a set of input/output procedures, which include the IFIP Input/Output Procedures [2], has been provided.

In the first chapter, a survey of the basic constituents and features of the language is given, and the formal notation, by which the syntactic structure is defined, is explained.

The second chapter lists all the basic symbols, and the syntactic units known as identifiers, numbers and strings are defined. Further, some important notions such as quantity and value are defined.

The third chapter explains the rules for forming expressions and the meaning of these expressions. Three different types of expressions exist: arithmetic, Boolean (logical) and designational.

The fourth chapter describes the operational units of the language, known as statements. The basic statements are: assignment statements (evaluation of a formula), goto statements (explicit break of the sequence of execution of statements), dummy statements, and procedure statements (call for execution of a closed process, defined by a procedure declaration). The formation of more complex structures, having statement character, is explained. These include: conditional statements, for statements (loops), compound statements, and blocks.

In the fifth chapter, the units known as declarations, serving for defining permanent properties of the units entering into a process described in the language, are defined.

The sixth chapter describes input/output procedures, for the transmission of data to and from an external medium.

There are three appendices which give further information. The first appendix describes the restrictions imposed by the System/360 Operating System ALGOL language on ALGOL 60 as described in the Revised ALGOL 60 Report [1].The second appendix describes the representation of basic symbols of ALGOL 60 in the 48- and 59-character sets accepted by the System/360 Operating System ALGOL compiler. The third appendix gives detailed examples of the language.

1. STRUCTURE OF THE LANGUAGE

The Revised ALGOL 60 Report introduces three different kinds of representations of the language. These are the reference language, the publication language and the hardware language representations. The latter give the representations of the language within the framework of the physical character sets available in various installations. All objects defined within the reference language are represented by a given set of symbols, and the hardware representations may differ from this set only in the choice of symbols.

The System/360 Operating System ALGOL compiler allows for two different sets of characters:

1. The ISO/DIN Proposal for the Representation of ALGOL Symbols on 80-column punched cards [3] (48-character set, based on H-version of the IBM card codes).

2. An extension of this proposal utilizing the syntactical characters of the Extended BCD Interchange Code (59-character set).

The characters available in (1) are a proper subset of (2). Both representations and the rules for transliterating them from the reference language are given in Appendix 2.

The description of the language in the following sections is given in terms of the first level (48-character set) of these hardware 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.

To show the flow of computational processes, certain nonarithmetic statements and statement clauses are added which may describe, e.g., alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these statements that one statement refer to another, statements may be provided with labels. A sequence of statements may be enclosed between the statement brackets 'BEGIN' and 'END' to form a compound statement.

Statements are supported by declarations which are not themselves computing instructions 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 sequence of declarations followed by a sequence of statements and enclosed between 'BEGIN' and 'END' constitutes a block. Every declaration appears in a block in this way and is valid only for that block.

A program is a block or compound statement which is not contained within another statement and which makes no use of other statements not contained logically within it.

Note: A program may contain declarations of procedures, called code procedures (c f. 5.4.6.), whose procedure bodies consist of any kind of code not contained physically within the program; however, this code is thought to be logically contained within the program.

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. Their interpretation is best explained by an example:

<ab> ::=/!*!<ab>/!<ab> <d>

Sequences of characters enclosed in the brackets < > represent metalinguistic variables whose values are sequences 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 sequence 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

---

1) Whenever the outcome of a certain process is left undefined or said to be undefined, this is to be interpreted in the sense that a program in which such a process is executed does not fully define a computational process.

The example chosen demonstrates that a metalinguistic formula does not define any meaning of a metalinguistic variable. In order to facilitate the study, however, the symbols used for distinguishing the metalinguistic variables (i. e. the sequences of characters appearing within the brackets < > as ab in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In additon, some formulae have been given in more than one place. Within syntactic descriptions the following definition will sometimes be used:

<empty> ::=

(i. e. no symbol)

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

The hardware representation of the 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

Letters do not have individual meaning. They are used to form identifiers and strings (c f. sections 2.4. and 2.6.).

2.2.1. Digits

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

Digits are used to form 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> :: = + ! - ! * ! / ! '/' ! 'POWER'

<relational operator> :: = 'LESS' ! 'NOTGREATER'! 'EQUAL' ! 'NOTLESS' !
'GREATER' ! 'NOT EQUAL'

<logical operator> :: = 'EQUIV' ! 'IMPL' ! 'OR' ! 'AND' ! 'NOT' !

<sequential operator> :: = 'GOTO' ! 'IF' ! 'THEN' ! 'ELSE' ! 'FOR' ! 'DO'

<separator> :: = , ! . ! ' ! .. ! ., ! . = ! 'STEP' ! 'UNTIL' ! 'WHILE' ! 'COMMENT'

<bracket> :: = (!) ! (/ ! /) ! '(' ! ')' ! 'BEGIN' ! 'END'

<declarator> :: = 'BOOLEAN' ! 'INTEGER' ! 'REAL' ! 'ARRAY' ! 'SWITCH'!
'PROCEDURE' ! 'CODE'

<specificator> :: = 'STRING' ! 'LABEL' ! 'VALUE'

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 between characters outside of strings (cf. 2.6.) or change to a new line have no significance for the representation of the numerical process itself (e.g., 'TRUE' and 'bTbRbUbEb' denote the same thing, hereby b denoting a blank space).

For the purpose of including text among the symbols of a program the following "comment" conventions hold:

The sequence of symbols:	is equivalent to:
.,'COMMENT'<any sequence not containing.,> .,	.,
'BEGIN' 'COMMENT' <any sequence not containing.,> .,	'BEGIN'
'END' <any sequence not containing 'END' or ., or 'ELSE'>	'END'

Equivalence is here meant that any of the three structures shown in the left-hand column may be replaced, in any occurrence outside of strings, by the basic symbol shown on the same line in the right-hand column without any effect on the action of the program. It is further understood that the comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

Note: The sequence of symbols constituting the text of the comment may consist of any available symbols and is not restricted to only the basic symbols described above (c f. symbols allowed in strings, section 2.6.3.). Blank spaces interspersed in the delimiting basic symbol ., or 'END' or 'ELSE' are ignored: e. g., the sequence . b, as well as the sequence ., is considered a delimiting basic symbol.

Example

The sequence

```
., 'COMMENT'bTHEbLASTbSTATEMENTbISbEXECUTEDbONLY, b
IFbANbERRORbOCCURS!b. b, b'END'OFbINNERbLOOP'END'OFb
OUTERbLOOP.,
```

is equivalent to the sequence

```
.,'END' 'END'.,
```

2.4 Identifiers

2.4.1. Syntax

<identifier> :: = <letter> ! <identifier> <letter> ! <identifier> <digit>

2.4.2 Examples

```
Q
SOUP
V17 A
A34KTbMNS
MARILYN
```

The following examples demonstrate sequences of symbols that are not correct identifiers:

1A	(starting digit)
A. L. ROBERTS	(. not allowed)
INPUT/OUTPUT	(/ not allowed)
NON-STOP	(- not allowed)

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; but there is no effective distinction between two different identifiers, when the first six basic symbols are common.

Apart from this rule, two different identifiers cannot be used to denote the same quantity. The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (c f. section 2.7. Quantities, Kinds and Scopes, and section 5. Declarations).

Note: According to the rule stated above, the identifiers

PROGRAM
PROGRAMMER
PROGRAM 15
PROGRA

are considered identical. Consequently only one of these identifiers denoting the same quantitity may be declared.

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> :: = '<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'-02
0177	+07.43'8	-'7
.5384	9.34'+10	'-4
+0.7300	2'-4	+'+5

Examples of invalid numbers are:

-0.	(decimal point not followed by a digit)
23+5	(delimiter ' is missing)
+7.'4	(decimal point not followed by a digit)
825.78E-5	(E instead of ')
-7.4'(5-3)	(Exponent part is an expression)

2.5.3 Semantics

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10. It is denoted by the separator'.

Notes: 2'-4 has the value +0.0002 and -'7 the value -10 000 000. Non-significant zeros, as in some of the examples of section 2.5.2, are allowed.

2.5.4 Types

Integers are of type 'INTEGER'. All other numbers are of type 'REAL' (c f. section 5.1. Type Declarations).

2.5.5 Range of Numbers

The numbers must be confined to the ranges described in section 2.8.

2.6. Strings

2.6.1 Syntax

<proper string> :: = <any sequence of symbols not containing the triplets '('or')'> ! <empty>

<open string> :: = <proper string> ! '(' < open string> ')' ! <open string> <open string>

<string> :: = '(' <open string> ')'

2.6.2 Examples

```
'(' . .bTHISbISbAb'('STRING')' ')'
'(' 5k, , -'('%%%'('&=/:')'Tt')'')'
```

2.6.3 Semantics

In order to enable the language to handle arbitrary sequences of symbols the string quotes '('and')' are introduced. There are 256 different symbols allowed within strings, including upper and lower case alphabetic characters, numerical

and syntactical characters, a blank space and other available characters, depending on the particular character set provided for use by the respective input/output devices. The symbol b denotes a blank space. It has no significance outside strings.

Strings are only used as a actual parameters of procedures (cf. sections 3.2. Function Designators, 4.7. Procedure Statements, and 6. Input/Output Procedures).

Notes: String quotes are exactly the sequences of symbols '(' or ')' and may not contain interspersed blank space..
A proper string may contain any sequence of symbols, except the sequences '(' and')' without interspersed blanks. The sequences '(' and ')' with interspersed blanks, e.g., 'b)b', can be used in a proper string.

Example:

The sequence

'('b'b(b'b')'

is a complete string, since the open string b'b(b'b does not contain a string quote, while the sequence

'('b'('b')'

is not a complete string since it contains two left, but only one right string quote.

Each symbol ' belongs to only one string quote and the string quotes are recognized from left to right.

Examples: The sequence '(')' represents a left string quote followed by the two symbols)'. The sequence '('(')' is a complete string consisting of a left and a right string quote enclosing the open string(.

2.7. Quantities, Kinds, and Scopes

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

The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid (c f.4.1.3.).

2.8 Values and Types

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.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (c f. section 3.1.4.1.).

The various "types" ('INTEGER', 'REAL', 'BOOLEAN') basically denote properties of values. The types associated with syntactic units refer to the values of these units.

A syntactic unit of type 'INTEGER' may have as its value an integer, I, within the following range:

$$-2^{31} = -2\ 147\ 483\ 648 \leq I \leq +2\ 147\ 483\ 647 = +2^{31}-1.$$

A syntactic unit of type 'REAL' may have as its value a real number, the modulus (absolute value) R of which lies within the following range:

$$16^{-65} \approx 2.4*10^{-78} \leq R \leq 7.2*10^{75} (1-16^{-14})*16^{63} \text{ or } R = 0.$$

A syntactic unit of type 'BOOLEAN' may have one of the two values:

'TRUE' or 'FALSE'.

Syntactic units of type 'REAL' are calculated with up to 17 (long form) or 8 (short form) significant decimal digits. [2] They are to be interpreted in the sense of numerical analysis, i.e., as entities defined inherently with only a finite accuracy. Therefore, the possibility of the occurence of a finite deviation from the mathematically defined result in any calculation involving syntactic units of type 'REAL' is explicitly understood. The control of the possible consequences of such deviations 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.

---

[2] The programmer may specify before translating an ALGOL program whether the precision of type 'REAL' calculations throughout the program is 7 to 8 or 16 to 17 significant decimal digits.

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, function designators, labels, switch designators, and elementary arithmetic, relational, logical, and sequential operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

<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
A 17
Q (/7, 2/)
X(/SIN(N*PI/2), Q(/3, N, 4/)/)
```

3.1.3. Semantics

A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2.). The type of the value of a particluar 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).

3.1.4. Subscripts

3.1.4.1. Subscripted variables designate values which are components of multi-dimensional 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. A subscript list may consist of up to 16 subscripts. The complete list of subscripts is enclosed in the subscript brackets (/ and /). 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. Each subscript position acts like a variable of type 'INTEGER' and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (cf. section 4.2.4.). 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).

Notes: Subscripts within a subscript list are evaluated from left to right. The subscript expressions may, of course, be nested, as demonstrated by the last example in section 3.1.2.

Examples: The fourth example in section 3.1.2. is a variable in a two-dimensional array Q. Its location in the array is specified by the first subscript 7 and the second subscript 2.

The fifth example is a variable in a two-dimensional array X. Its location in the array is determined in the following way: The current values of N and PI are used to evalute SIN (N*PI/2). The value of this expression, transferred to type 'INTEGER' (c f. 4.2.4.), yields the first subscript of X. Then the value of the variable in three-dimensional array Q identified by the subscripts 3, current value of N (transferred to type 'INTEGER'), yields the second subscript of X. If, for example, at the time the subscripted variable X (/SIN(N*PI/2), Q(/3, N, 4/)/) is used, N = 1, PI = 3.14 ..., and Q(/3, 1, 4/)= 0. 2, then the variable X(/1, 0/) in array X is to be taken.

3.2. 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>)

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

3.2.2. Examples

```
SIN (A - B)
J (V + S, N)
R
S(X - 5) TEMPERATURE .. (T) PRESSURE .. (P)
COMPILE ('('.='')') STACK .. (Q)
```

3.2.3. Semantics

A function designator defines a single numerical or logical value, which results through the application of a given set of rules defined by a procedure declaration (cf. 5.4 Procedure Declaration, especially 5.4.4.) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7. Procedure Statements. Not every procedure declaration defines the value of a function designator.

3.2.4. Standard Functions

Certain identifiers are used for standard functions that are expressed as procedures. These functions are available without explicit declaration. They are considered as declared in a block surrounding the whole program. If an identifier representing a standard function is explicitly declared in any block, this declaration is to be regarded as a redeclaration of the identifier; within this block the identifier assumes the significance implied by the nature of the declaration given and the standard function represented by the identifier is inaccessible (cf. 4.1.3. and 5.). The list of standard functions consist of ten identifiers.

3.2.4.1. Standard Functions of Analysis

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 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 values of type 'REAL', except for SIGN (E) which will have values of type 'INTEGER'.

Notes: Each of the standard functions is defined only if the values of both argument and function designator lie within the ranges described in section 2.8., e.g., $E \leq 0$ for SQRT (E) or $E \leq$ LN (MAX)$\approx$174 for EXP (E), where MAX = $(1 - 16^{-14}) * 16^{63}$ is the maximum 'REAL' type value. Beyond that the argument E of SIN and COS is restricted by the condition

$$-3.6*10^{15} \approx -\pi*2^{50} < E < +\pi*2^{50} \approx +3.6*10^{15}$$

in case of long form of 'REAL' type values and by the condition

$$-8.2*10^{5} \approx -\pi*2^{18} < E < +\pi*2^{18} \approx +8.2*10^{5}$$

in case of short form of 'REAL' type values (c f. footnote 2) of 2.8.).

The assumed implicit declarations of the above standard functions might be (c f. 5.4.):

```
'REAL' 'PROCEDURE' ABS (E)., 'VALUE' E., 'REAL' E.,
   ABS.='IF'E'NOTLESS'0'THEN'E'ELSE'-E.,

'INTEGER''PROCEDURE'SIGN(E).,'VALUE'E.,'REAL'E.,
   SIGN.='IF'E'GREATER'0'THEN'+1
                'ELSE''IF'E'EQUAL'0'THEN'0
                'ELSE'-1.,

 'REAL''PROCEDURE'SQRT(E).,'VALUE'E.,REAL'E.,
  <procedure body>.,

 'REAL''PROCEDURE'SIN(E).,'VALUE'E.,'REAL'E.,
  <procedure body>.,
```

etc.

3.2.4.2. Transfer Function

A further standard function is the transfer function

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.

Notes: The assumed implicit declaration of ENTIER might be:

```
'INTEGER''PROCEDURE'ENTIER(E).,'VALUE'E.,'REAL'E.,'CODE'
```

Examples

```
ENTIER    (2.9)  =   2
ENTIER  (- 2.9)  =  -3
ENTIER    (5.0)  =   5
ENTIER    (0.0)  =   0
```

3.2.4.3. Length of a String

Finally there exists the standard function

LENGTH(S)

which operates on a string argument S and yields an 'INTEGER' type value, namely the number of symbols of the open string enclosed between the outermost string quotes of S.

Notes: If the open string is empty, LENGTH yields the value 0.
The assumed implicit declaration of LENGTH might be

```
'INTEGER''PROCEDURE'LENGTH(S).,'STRING'S.,'CODE'
```

Examples

```
LENGTH('('bABCb')')=5
LENGTH('(''BEGIN'')')=7
LENGTH('('PRICE=5$')')=8
LENGTH('('')')=0
LENGTH('(''('bb')'')')=8
```

3.3. Arithmetic Expressions

3.3.1. Syntax

```
<adding operator> :: = + ! -
<multiplying operator> ::= * ! / ! '/'
<primary> ::= <unsigned number> ! <variable> !
              <function designator> ! (<arithmetic expression>)
<factor> ::= <primary> ! <factor> 'POWER' <primary>
<term> ::= <factor> ! <term> >multiplying operator> <factor>
<simple arithmetic expression> :: = <term> !
                                    <adding operator> <term>!
                                    <simple arithmetic expression>
                                    <adding operator> <term>
<if clause> :: = 'IF' <Boolean expression>'THEN'
<arithmetic expression> ::= <simple arithmetic expression> !
                            <if clause> <simple arithmetic expression> 'ELSE'
                            <arithmetic expression>
```

3.3.2 Examples

Primaries:

```
U
OMEGA
SUM
COS(Y+Z*3)
7.394'-8
W(/I+2, 8/)
(A-3/Y+VU'POWER'8)
```

Factors:

```
U
OMEGA
SUM'POWER'COS(Y+Z*3)
7.394'-8'POWER'W(/I+2, 8/)'POWER'(A-3/Y+VU'POWER'8)
```

Terms:

```
U
OMEGA*SUM'POWER'COS(Y+Z*3)/7.394'-8'POWER'W(/I+2, 8/)
      'POWER'(A-3/Y+VU'POWER'8)
```

Simple arithmetic expressions:

```
U+OMEGA*SUM'POWER'COS(Y+Z*3)/7.394'-8'POWER'
      W(/I+2, 8/)'POWER'(A-3/Y+VU'POWER'8)
-5.0
```

Arithmetic expressions:

```
W*U-Q(S+CU)'POWER'2
'IF'Q'GREATER'0'THEN'S+3*P/A'ELSE'2*S+3*Q
'IF'A'LESS'0'THEN'U+V'ELSE''IF'A*B'GREATER'17
   'THEN'U/V'ELSE''IF'K'NOT EQUAL'Y'THEN'V/U'ELSE'0
A*SIN(OMEGA*T)
0.57'12*A(/N*(N-1)/2, 0/)
(A*ARCTAN(Y)+Z)'POWER'(7+Q)
'IF'Q'THEN'N-1'ELSE'N
'IF'A'LESS'0'THEN'A/B'ELSE''IF'B'EQUAL'0
   'THEN'B/A'ELSE'Z
```

Note: The examples of primaries, factors, terms are the constituents of the first example of a simple arithmetic expression and demonstrate the successive construction of simple arithmetic expressions.

Examples of incorrect arithmetic expressions:

(A+3'5'POWER'(N+1)	(right parenthesis missing)
-3*-5	(-5 is not a factor)
(A+B)'POWER'-0.5	(exponent is not a primary)
X+Y+('IF'A'EQUAL'0'THEN'1)	(alternative missing)
X+'IF'A'EQUAL'0'THEN'1'ELSE'0	(second summand is not a term, parentheses missing)

3.3.3. Semantics

An arithmetic expression is a rule for computing a numerical value (cf. 2.8.). 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 (cf. 2.5.). For variables it is the current value (assigned last in the dynamic sense (cf 4.2.)), and for function designators it is the value arising from the computing rules defining the procedure (cf. 5.4.4. Values of Function Designators) 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 Boolean expressions (cf. 3.4.). In this case, according to the syntax, the form of the arithmetic expression is

'IF'B'THEN'A1'ELSE'A2

where B is a Boolean expression, Al a simple arithmetic expression and A2 an arithmetic expression. The selection of a simple arithmetic expression is made as follows: The Boolean expression B is evaluated; if it has the value 'TRUE' then the value of the whole expression (i.e. of 'IF'B'THEN'Al'ELSE' A2) is the value of the first (simple) arithmetic expression A1; if B has the value 'FALSE' then the value of the whole expression is the value of the second arithmetic expression A2, which has to be evaluated by a recursive analysis in the same way if it is not a simple arithmetic expression.

Example. The last correct example of section 3.3.2 is an arithmetic expression whose value is the value of one of the three simple arithmetic expressions A/B, B/A, Z depending on the current values of the Boolean expressions A'LESS' 0, B'LESS'0; namely:

if $A < 0$, then the value of A/B is selected,
if $A \geq 0$ and $B = 0$, then the value B/A is selected,
if $A \geq 0$ and $B \neq 0$, then the value of Z is selected.

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 a set of rules, given in sections 3.3.4.1 to 3.3.4.3. below. However, if the type of an arithmetic expression according to the rules cannot be determined without evaluating an expression or ascertaining the type or value of an actual parameter, it is 'REAL'.

Examples. Assume I and J of type 'INTEGER' and A of type 'REAL'. Then the values of the expressions

I'POWER'J (without regard to the sign of J),
'IF'B'THEN'J'ELSE'A (without regard to the value of B)

are of type 'REAL'.

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 <terms> / <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

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

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' mathematically defined as follows:

A'/'B=SIGN(A/B)*ENTIER(ABS(A/B))

(cf. section 3.2.4.).

Examples

10'/'5 = 2,
9'/'5 = 1,
(-9)'/'5 = -1,
9'/'(-5) = -1.

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

2'POWER'N'POWER'K means $(2^N)^K$

while

2'POWER'(N'POWER'M) means $2^{(N^M)}$.

The resulting value is of type 'INTEGER' if the base is of type 'INTEGER' and the exponent is an unsigend integer (c f. 2.5.) (e.g. 3'POWER'7, I'POWER'0, (I+J) 'POWER'5

but not I'POWER'J or I'POWER' (3+0), where I, J are variables of type 'INTEGER'. In all other cases it is of type 'REAL'. Besides this rule concerning the type, the numerical value of

B'POWER'E

is given by the rules stated in the following tables:

E of type 'INTEGER':

conditions	result
E>0	B*B*...*B (E times)
E=0, B≠0	1
E≤0, B=0	undefined
E<0, B≠0	1/(B*B*...*B) (the denominator has -E factors)

E of type 'REAL':

conditions	result
B>0	EXP(E*LN(B))
B=0, E>0	0
B=0, E≤0	undefined
B<0	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:	'POWER'
second:	*, /, '/'
third:	+, -

3.3.5.2. The expression between a left parenthesis and the matching right parathesis 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.

Examples.

A/B*C	means	(A/B)*C
-X'POWER'Y	means	-(X'POWER'Y)
-3'POWER'0.5	means	-(3'POWER'0.5)
-9'/'5	means	-(9'/'5)
(-9)'/'5	means	(-9)'/'5

3.4. Boolean Expressions

3.4.1. Syntax

```
<relational operator> ::= 'LESS' ! 'NOTGREATER' ! 'EQUAL' !
                          'NOTLESS' ! 'GREATER' ! 'NOTEQUAL'
<relation> ::= <simple arithmetic expression>
               <relational operator> <simple arithmetic expression>
<Boolean primary> ::= <logical value> ! <variable> !
                      <function designator> ! <relation> ! (<Boolean expression>)
<Boolean secondary> ::= <Boolean primary> ! 'NOT' <Boolean primary>
<Boolean factor> ::= <Boolean secondary>!
                     <Boolean factor> 'AND' <Boolean secondary>
<Boolean term> ::= <Boolean factor> ! <Boolean term>
                   'OR' <Boolean factor>
<implication> ::= <Boolean term> ! <implication> 'IMPL' <Boolean term>
<simple Boolean> ::= <implication> !
                     <simple Boolean> 'EQUIV' <implication>
<Boolean expression> ::= <simple Boolean> !
                         <if clause> <simple Boolean> 'ELSE'
                         <Boolean expression>
```

3.4.2. Examples

```
x'EQUAL'-2
Y'GREATER'V'OR'Z'LESS'Q
A+B'GREATER'-5'AND'Z-D'GREATER'Q'POWER'2
P'AND'Q'OR'X'NOTEQUAL'Y
Q'EQUIV''NOT'A'AND'B'AND''NOT'C'OR'D'OR'E'IMPL''NOT'F
'IF'K'LESS'1'THEN'S'GREATER'W'ELSE'H'NOTGREATER'C
'IF''IF''IF'A'THEN'B'ELSE'C'THEN'D'ELSE'F'THEN'G'ELSE'H'LESS'K
```

The following example of a Boolean expression is incorrect because a relation may contain only simple arithmetic expressions:

'IF'B'THEN'0'ELSE'1'LESS'N

To be correct it has to be written:

('IF'B'THEN'0'ELSE'1')'LESS'N

3.4.3. Semantics

A Boolean expression is a rule for computing a logical value (cf. 2.8.). The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.4.4. Types

Variables and function designators entered as Boolean primaries must be declared 'BOOLEAN' (cf. section 5.1. Type Declarations and section 5.4.4. Values of Function Designators).

3.4.5. The Operators

Relations take on the value 'TRUE' whenever the corresponding relation is satisfied for the actual values of the expressions involved (cf. 3.3.3.), otherwise 'FALSE'.

Examples. The value of

5'EQUAL'3	is	'FALSE'
4'EQUAL'4.0'0	is	'TRUE'
N+1 'LESS' 0	is	'TRUE', if, for example, the current value of N is - 5.25.

Note: If the values of the two arithmetic expressions involved are of different type, they are both converted to type 'REAL' before evaluating the relation.

The meaning of the logical operators 'NOT', 'AND', 'OR', 'IMPL' (implies) and 'EQUIV' (equivalent) is given by the following function table.

B1	'FALSE'	'FALSE'	'TRUE'	'TRUE'
B2	'FALSE'	'TRUE'	'FALSE'	'TRUE'
'NOT'B1	'TRUE'	'TRUE'	'FALSE'	'FALSE'
B1'AND'B2	'FALSE'	'FALSE'	'FALSE'	'TRUE'
B1'OR'B2	'FALSE'	'TRUE'	'TRUE'	'TRUE'
B1'IMPL'B2	'TRUE'	'TRUE'	'FALSE'	'TRUE'
B1'EQUIV'B2	'TRUE'	'FALSE'	'FALSE'	'TRUE'

Examples

3'LESS'0'OR'4GREATER'0	has the value 'TRUE'
X'LESS'1'AND'X'NOTLESS'1	has always the value 'FALSE'
X'LESS'1'OR'X'NOTLESS'1	has always the value 'TRUE'
'TRUE''OR'B	has always the value 'TRUE'
'FALSE''AND'B	has always the value 'FALSE'
A'AND''NOT'A	has always the value 'FALSE'
A'OR''NOT'A	has always the value '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 expression according to section 3.3.5.
second:	'LESS', 'NOTGREATER', 'EQUAL', 'NOTLESS', 'GREATER', 'NOTEQUAL'
third:	'NOT'
fourth:	'AND'
fifth:	'OR'
sixth:	'IMPL'
seventh:	'EQUIV'

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

Examples. The second, third, fourth and seventh example of section 3.4.2 are to be interpreted as follows:

```
(Y'GREATER'V)'OR'(Z'LESS'Q),

((A+B)'GREATER'(-5))'AND'((Z-D)'GREATER'(Q' POWER'2)),

(P'AND'Q)'OR'(X'NOTEQUAL'Y),

'IF'('IF'('IF'A'THEN'B'ELSE'C)'THEN'D'ELSE'F)
                                'THEN'G'ELSE'(H'LESS'K).
```

Assuming that in the fourth example, the variables have the values:

```
P='TRUE', Q='FALSE', X=-5.7'+4,  Y=0,
```

then P'AND'Q has the value 'FALSE', the relationX'NOTEQUAL'Y has the value 'TRUE' and the whole expression has the value of 'FALSE''OR''TRUE', i.e. the value 'TRUE'.

3.5. Designational Expressions

3.5.1. Syntax

```
<label> ::= <identifier>
<switch identifier> ::= <identifier>
<switch designator> ::= <switch identifier> (/<subscript expression>/)
<simple designational expression> ::= <label> ! <switch designator> !
                                      (<designational expression>)
<designational expression>::= <simple designational expression> !
                              <if clause> <simple designational expression>
                              'ELSE' <designational expression>
```

3.5.2. Examples

```
P9
CHOOSE (/N-1/)
TOWN(/'IF'Y'LESS'0'THEN'N'ELSE'N+1/)
'IF'AB'LESS'C'THEN'P'ELSE'Q(/'IF'W'NOTGREATER'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 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 expressions thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.5.4. The Subscript Expression

The evaluation of the subscript expression is analogous to that of subscripted variables (cf. section 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values 1, 2, 3 ..., n, where n is the number of entries in the switch list (cf. 5.3.).

Examples. The value of the first example of section 3.5.2. simply is the label P9.

The value of the second example is the value of the (N-1)st designational expression of the switch list in the declaration of the switch CHOOSE. If, for example, at the time the designational expression is to be evaluated, N has the value 2.2, then N-1 has to be transferred to the 'INTEGER' type value 1 and therefore the value of the designational expression is the value of the first designational expression within the switch list of CHOOSE.

The last example is to be interpreted as

```
'IF'(AB'LESS'C)'THEN'P'ELSE'Q(/'IF'(W'NOTGREATER'0)'THEN'2
                                                    'ELSE'N/)
```

Assuming AB=0, C=1, then the value of this expression is the label P; but, assuming AB=0, C=0, then the value of the expression depends on the values of W and N. If W=4, N=5.2, then the fifth designational expression of the switch list of Q is to be evaluated.

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 goto statements, which define their successor explicitely, and shortened 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

4.1.1. Syntax

```
<unlabelled basic statement> ::= <assignment statement> !
                                 <goto statement> ! <dummy statement>
                                 <procedure statement>
<basic statement> ::= <unlabelled basic statement> !
                      <label> . . <basic statement>
<unconditional statement> ::= <basic statement> !
                              <compound statement> ! <block>
<statement> ::= <unconditional statement> !
                <conditional statement> ! <for statement>
<compound tail> ::= <statement> 'END' !
                    <statement>., <compound tail>
<block head> ::= 'BEGIN' <declaration> !
                 <block head> ., <declaration>
<unlabelled compound> ::= 'BEGIN' <compound tail>
<unlabelled block> ::= <block head>.,<compound tail>
<compound statement> ::= <unlabelled compound> !
                         <label> .. >compound statement>
<block> ::= <unlabelled block> ! <label> .. <block>
<program> ::= <block> ! <compound statement>
```

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound statement:

```
L..L.. ...'BEGIN'S.,S., ...S.,S'END'
```

Block:

```
L..L.. ...'BEGIN'D.,D., ...D.,S.,S., ...S.,S'END'
```

It should be kept in mind that each of the statements S may again be compound statement or block.

4.1.2 Examples

Basic statements:

```
A.=P+Q
'GOTO'NAPLES
START..CONTINUE..W.=7.993
```

Compound statement:

```
'BEGIN'X. =0.,'FOR'Y. = 1'STEP'1'UNTIL'N'DO'
        X. =X+A(/Y/).,
        'IF'X'GREATER'Q'THEN''GOTO'STOP
        'ELSE''IF'X'GREATER'W-2'THEN''GOTO!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'BLOCKQ
```

4.1.3. Semantics

The definitions of compound statement and block allow a sequence of statements to be grouped within the statement brackets 'BEGIN' and 'END' and handled as one statement. A compound statement or a block may appear in any syntactical context where a statement is permitted. Thus, by means of a compound statement or block, any actions to be executed on a statement (e. g. by for statements or conditional statements) can also be executed on sequences of statements.

Additionally, 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 (c.f. 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 block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be nonlocal to it, i. e., will represent the same entity inside the block and in the level immediately outside it. A label separated by . . from a statement, i. e., labelling that statement, behaves as though declared in the head of the smallest embracing block, i. e., the smallest block whose brackets 'BEGIN' and 'END' enclose that statement. In this context, a procedure body or a program must be considered as if it were enclosed by 'BEGIN' and 'END' and treated as a block.

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

Examples. The following example

```
'BEGIN''REAL'A, B, C.,
        :
        :
        D..
        :
        :
        'BEGIN''INTEGER'A, E.,
                :
                :

                        :
                        :
                    'BEGIN'A.=B.,
                              :
                              :
                            C..
                              :
                    'END'     :
                'END'         :
                              :
'END'
```

consists of two nested blocks, the inner one containing a compound statement. There are declared, explicitly or by occurrence as labels, the following seven identifiers: two 'REAL' type variables A and C whose scope is the outer block except the inner block, a 'REAL' type variable B and a label D whose scope is the whole outer block including the inner one, two 'INTEGER' type variables A and E and a label C whose scope is only the inner block. It should be noted that the scope of the label C, which occurs within the compound statement included by the innermost statement brackets 'BEGIN' and 'END', is not restricted to this compound statement, but is the whole inner block.

In the example

```
'PROCEDURE'P(X).,'REAL'X.,
    L..'IF'X'GREATER'1
            'THEN''BEGIN'X.=X-1.,'GOTO'L'END'
            'ELSE''IF'X'LESS'0
                'THEN''BEGIN'X.=X+1.,'GOTO 'L'END'
```

the scope of the label L is the procedure body, though there is no explicit block. A similar rule concerns a label labelling a whole program.

4.2. Assignment Statements

4.2.1. Syntax

<left part> ::= <variable> .= ! <procedure identifier> .=
<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'GREATER'Y'AND'Z
```

The following are incorrect assignment statements:

A+B.=C	(left part is an expression)
'TRUE'.=U'OR'V	(left part is not a variable)
F(X).=X+1	(left part is a function designator)
X(/5/).=A+U'OR'V	(right part is neither arithmetic nor Boolean)

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of the procedure defining the value of a function designator (cf. section 5.4.4.). 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 (cf. 3.1.4.).

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

The type associated with all variables and procedure identifiers of a left part list must be the same. If this type is 'BOOLEAN', the expression must likewise be 'BOOLEAN'. If the type is 'REAL' or 'INTEGER', the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are automatically invoked. For transfer from 'REAL' to 'INTEGER' type, the transfer function is understood to yield a result equivalent to

ENTIER (E+0.5)

where E is the value of the expression . The type associated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (c f. section 5.4.4.).

Examples. Assume B is declared 'BOOLEAN', I, J, K, L, M are declared 'INTEGER', X, Y are declared 'REAL' and A is declared to be an 'INTEGER' type array. Then the following assignment statements are incorrect:

```
B.=X+J          (wrong type of left part)
X.=B            (wrong type of left part)
X.=I.=Y+J       (different types of left parts).
```

The assignment statements

```
I.=1.8., J.=-1.8., K.=1.5., L.=-1.5., M.=1.2.,
```

assign the values 2, -2, 2, -1, 1 to the variables I, J, K, L, M. If I has the value 2 then either of the following statements

```
I.=A(/I/).=I+1.5
```

or

```
A(/I/).=I.=I+1.5
```

assigns the value 4 both to I and A(/2/), but does not influence the value of A(/4/).

4.3. Goto Statements

4.3.1. Syntax

<goto statement> ::= 'GOTO' <designational expression>

4.3.2. Examples

```
'GOTO'P9
'GOTO'CHOOSE(/N-1/)
'GOTO'TOWN(/'IF'Y'LESS'0'THEN'N'ELSE'N+1/)
'GOTO''IF'AB'LESS'C'THEN'B'ELSE'Q(/'IF'W'LESS'0
    'THEN'2'ELSE'N/)
```

4.3.3. Semantics

A goto 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 (cf. 3.5.). Thus the next statement to be executed will be the one having this value as its label. The action of a goto statement is defined only if the value of the designational expression is defined by the rules of section 3.5.

4.3.4. Restriction

Since labels are inherently local, no goto statement can lead from outside into a block or a procedure body. A goto statement may, however, lead from outside into a compound statement (cf. 4.1.3.).

Note: Concerning the additional action of a goto statement leaving a block see section 5.

Examples. The four statements of section 4.3.2. define the statements labelled by the values determined by the examples of section 3.5.2. to be their successors.

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 (especially before the delimiter 'END').

4.5. Conditional Statements

4.5.1. Syntax

```
<if clause> ::= 'IF' <Boolean expression> 'THEN'
<unconditional statement> ::= <basic statement> !
                              <compound statement> ! <block>
<if statement> ::= <if clause> <unconditional statement>
<conditional statement> ::= <if statement> ! <if statement> 'ELSE'
                            <statement> ! <if clause> <for statement> !
                            <label> .. <conditional statement>
```

4.5.2. Examples

```
'IF'X'GREATER'0'THEN'N.=N+1

'IF'V'GREATER'U'THEN'A..Q.=N+M'ELSE''GOTO'R

'IF'S' LESS'0'OR'P'NOTGREATER'Q'THEN'
      AA..'BEGIN''IF'Q1'LESS'V'THEN'A.=V/S
                    'ELSE'Y.=2*A' END'
          'ELSE''IF'V'GREATER'S'THEN'A.=V-Q1
                    'ELSE''IF'V'GREATER'S-1'THEN''GOTO'S1
```

Note: The following example is incorrect because the first 'THEN' is followed by a conditional statement:

```
'IF'A'LESS'B'THEN''IF'A'LESS'C'THEN'A.=B-C'ELSE'A. = C-B
```

The restriction that the statement following 'THEN' must be an unconditional statement, while the statement following 'ELSE' may be an arbitrary statement is necessary in order to avoid ambiguity. If a conditional statement follows the delimiter 'THEN' it cannot be determined to which of the nested if clauses the delimiter 'ELSE' corresponds. (In the above example, the delimiter 'ELSE' could correspond to the first or to the second 'THEN'). This ambiguity is avoided by enclosing the statement following the first 'THEN' by the brackets 'BEGIN' and 'END' and so forming an unconditional statement.
A similar ambiguity arises by the following example (cf. 4.6.):

```
'IF'B1'THEN''FOR'I.=1'STEP'1'UNTIL'N'DO'
     'IF'B2'THEN'A (/I/).=B(/I/)'ELSE'A(/I/).=0
```

To avoid the ambiguity as to which if clause the delimiter 'ELSE' corresponds, a for statement following 'THEN' must not be followed by an 'ELSE'; therefore a delimiter 'ELSE' corresponds always to an if clause within the for statement. If the for statement is to be followed by 'ELSE', the for statement must be enclosed in the brackets 'BEGIN' and 'END'.

4.5.3. Semantics

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

According to the syntax, three different forms of conditional statements are possible (regarding a labelled conditional statement as belonging to one of the three other cases). These forms may be illustrated as follows:

(1) 'IF'B'THEN'SU.,SN

(2) 'IF'B'THEN'SF.,SN

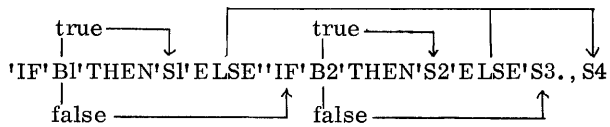
(3) 'IF'B'THEN'SU'ELSE'S.,SN

Here B is a Boolean expression, SU an unconditional statement, SF a for statement, S an arbitrary statement and SN the next statement following the complete conditional statement.

In all three forms, the Boolean expression B is evaluated. If its value is 'TRUE' the statement following the 'THEN' (i.e. SU or SF) is executed. Unless this statement explicitly defines its successor (i. e. by a goto statement, cf. 4.3.), the next statement executed will be SN (thus skipping the statement S following the 'ELSE' in form 3).

If the value of the Boolean expression is 'FALSE', then in the first two forms the statement SU or SF following the 'THEN' is skipped, and SN is the next statement executed. In form 3, if the value of the Boolean expression is 'FALSE' the statement SU following the 'THEN' is skipped and the statement S following the 'ELSE' is executed instead. Since this statement again may be a conditional statement, this process might be applied recursively.

For further explanation the following diagram might be useful:



Note: Effectively, in any case the delimiter 'ELSE' defines that the successor of the statement, to which the delimiter 'ELSE' follows, is the statement following the complete conditional statement.

Example. The third example of section 4.5.2. has the following effect:

S'LESS'0'OR'P'NOTGREATER'Q

is evaluated according to the rules of section 3.4. If the value is 'TRUE' then the compound statement

'BEGIN''IF'Q1'LESS'V'THEN'A.=V/S
'ELSE'Y.=2*A'END'

is executed; i. e., the relation Q1'LESS'V is evaluated; and, if its value is 'TRUE', the assignment statement A.=V/S is executed; if its value is 'FALSE', the assignment statement Y.=2*A is executed. After that the rest of the complete conditional

statement is skipped.

If the value of the first Boolean expression is 'FALSE', then the compound statement is skipped and the statement following the first 'ELSE', i. e.

```
'IF'V'GREATER'S'THEN'A.=V-Q1
    'ELSE''IF'V'GREATER'S-1'THEN''GOTO'S1
```

is executed. This means that if the value of V'GREATER'S is 'TRUE', the assignment statement A.=V-Q1 is executed and the rest of the conditional statement is skipped. If the value of V'GREATER'S is 'FALSE', the assignment statement A.=V-Q1 is skipped and the statement 'IF'V'GREATER'S-1'THEN''GOTO'S1 is executed, i. e. the relation V'GREATER'S-1 evaluated and then the statement GOTO'S1 executed if the value is 'TRUE' or skipped if it is 'FALSE'.

4.5.4. Goto into a Conditional Statement

The effect of a goto statement leading into a conditional statement follows directly from the effect of 'ELSE' as explained in the note in section 4.5.3. The statement designated by the goto statement and its successors are executed until meeting an 'ELSE'. This 'ELSE' then effects the skipping of the rest of the conditional statement to which it belongs.

Example. If a goto statement leads to the label AA in the third example of section 4.5.2., the compound statement labelled by AA will be executed. The 'ELSE' following that compound statement causes the rest of the complete conditional statement to be skipped, and its successor to be executed next, just as if the compound statement had been entered because the preceding condition, S'LESS'0'OR'P'NOTGREATER'Q, was 'TRUE'.

4.6. For Statements

4.6.1. Syntax

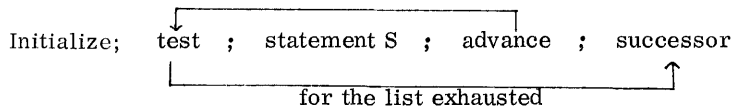
```
<for list element> ::= <arithmetic expression> !
                       <arithmetix expression> 'STEP' <arithmetic expression>
                        'UNTIL' <arithmetic expression> .
                       <arithmetic expression> 'WHILE' <Boolean expression>
<for list> ::= <for list element> ! <for list>, <for list element>
<for clause> ::= 'FOR' <variable> .= <for list> 'DO'
<for statement> ::= <for clause> <statement> !
                    <label> .. <for statement>
```

4.6.2. Examples

```
'FOR'I.=0, 1, -1'DO'A(/I/).=2*I
'FOR'Q.=1'STEP' S'UNTIL'N'DO'A(/Q/.=B(/Q)
'FOR'K.=1, K*2'WHILE'K'LESS'N'DO'
    'FOR'J.=I+G, L, 1'STEP' ' UNTIL'N, 2*K'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. In addition it performs a sequence of assignments to its controlled variable, i.e. the variable following the delimiter 'FOR'. The process may be visualized by means of the following diagram:



In this picture the word initialize means: perform the first assignment of the for clause. Advance means: perform the next assignment of the for clause. Test determines if the last assignment has been done. If so, the execution continues with the successor of the for statement. If not, the statement following the for clause is executed.

Note: The case may occur, too, that the first test, immediately following initialize, determines that the last assignment has already been done and the execution is to continue with the successor of the for statement. In this case the statement S is not executed at all.

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. A for list element which is an arithmetic expression E gives rise to one value to be assigned. The execution may be described in terms of additional ALGOL statements as follows:

```
V.=E.,
statement S.,
'GOTO' ELEMENT EXHAUSTED.,
```

where V is the controlled variable of the for clause and ELEMENT EXHAUSTED points to the action according to the next element in the for list, or, if the element just handled is the last of the for list, to the next statement in the program.

Example. The first example of section 4.6.2. is equivalent to the following sequence of statements:

```
I. =0., A(/I/). =2*I.,
I. =1., A(/I/). =2*I.,
I. =1., A(/I/). =2*I
```

4.6.4.2. Step-until-element. An element of 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:

```
'BEGIN' 'REAL' STEP, TESTVALUE.,
        V.=A.,
        STEP.=B., TESTVALUE.=C.,
   L2..'IF'(V-TESTVALUE)*SIGN (STEP) 'GREATER' 0
          'THEN' 'GOTO' ELEMENT EXHAUSTED.,
    statement S.,
         STEP.=B., TESTVALUE.=C.,
         V.=V+STEP.,
         'GOTO'L2
'END'
```

where the notation of V and ELEMENT EXHAUSTED is the same as in section 4.6.4.1. above; STEP and TESTVALUE are auxiliary introduced variables whose names have to be changed suitably if there is any conflict with another identifier STEP or TESTVALUE occuring in the for statement.

Examples. Assuming that S=2 and N=5 when entering the for statement of the second example of section 4.6.2, this statement is equivalent to:

```
Q.=1., A(/Q/)=B(/Q/).,
Q.=3., A(/Q/).=B(/Q/).,
Q.=5., A(/Q/).=B(/Q/)
```

If S=-1 and N=-1.5, then the statement is equivalent to:

```
Q.=1., A(/Q/).=B(/Q/).,
Q.=0., A(/Q/).=B(/Q/).,
Q.=-1., A(/Q/).=B(/Q/).,
```

Finally, if S=1 and N=0, the whole for statement would be skipped.

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''NOT'F'THEN''GOTO'ELEMENT EXHAUSTED.,
   statement S.,
  'GOTO'L3.,
```

where the notation is the same as in 4.6.4.1. above.

Example. Assuming N=4.1 when entering the for statement of the third example of section 4.6.2., this statement is equivalent to:

```
K.=1.,
'FOR'J.=I+G, L, 1'STEP'1'UNTIL'4.1, 2*K'DO'
                    A(/K, J/).=B(/K, J/).,

K.=2.,
'FOR'J.=I+G, L, 1'STEP'1'UNTIL'4.1, 2*K'DO'
                    A(/K, J/).=B(/K, J/).,

K.=4.,
'FOR'J.=I+G, L, 1'STEP'1'UNTIL'4.1, 2*K'DO'
                    A(/K, J/).=B(/K, J/)
```

Each of these three "inner" for statements is equivalent to:

```
J.=I+G., A(/K,J/).=B(/K,J/).,

J.=L.,   A(/K,J/).=B(/K,J/).,

J.=1.,   A(/K,J/).=B(/K,J/).,

J.=2.,   A(/K,J/).=B(/K,J/).,

J.=3.,   A(/K,J/).=B(/K,J/).,

J.=4.,   A(/K,J/).=B(/K,J/).,

J.=2*K., A(/K,J/).=B(/K,J/)
```

The meaning of the term "controlled variable" in the previous sections is obvious if the delimiter 'FOR' is followed by a simple variable: then it is this variable. If, on the other hand, the delimiter 'FOR' is followed by a subscripted variable, the identity of this variable is determined by evaluating the subscripts (c f. 3.1.4.) once when entering the for statement. The variable so determined is taken as the controlled variable throughout the for statement, even if the value of a variable in the subscript expressions is altered.

This might be explained by introducing in the following way auxiliary variables as the subscripts of the controlled variable:
The statement

```
'FOR' V(/I/).=<for list> 'DO' statement S.,
```

where I is an arithmetic expression, is to be executed as

```
'BEGIN' 'INTEGER' SUBSCRIPT.,
     SUBSCRIPT.=I.,
     'FOR' V(/SUBSCRIPT/).= <for list> 'DO' statement S
'END'
```

In this connection, the latter for statement is to be executed as explained above in section 4.6.4.1 through 4.6.4.3 with the exception that in these explanations V is to be replaced by V(/SUBSCRIPT/). If the controlled variable possesses more than one subscript, an analogous rule is valid. Of course, the identifiers of the auxiliary variables (SUBSCRIPT) have to be chosen so that no conflicts with other identifiers arise.

Example. The statements

```
I.=0
'FOR'V(/I/).=1'STEP'1'UNTIL'3'DO'
      'BEGIN'I.=I+2., V(/I/).=I'END'
```

are to be interpreted as:

I.=0.,
V(/0/).=1., I.=2., V(/2/).=2.,
V(/0/).=2., I.=4., V(/4/).=4.,
V(/0/).=3., I.=6., V(/6/).=6

4.6.5. The Value of the Controlled Variable upon Exit

Upon exit out of the statement S (supposed to be compound) through a goto statement the value of the controlled variable will be the same as it was immediately preceding the execution of the goto statement.

If on the other hand, the exit is due to exhaustion of the for list as described above, i. e. by one of the above statements 'GOTO' ELEMENT EXHAUSTED, the value of the controlled variable is undefined after the exit.

4.6.6. Goto Leading into a For Statement

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

Note: This rule applies even if the goto statement occurs in a procedure body (cf. 5.4.) outside the for statement and the procedure is activated inside the for statement. So, in the following example the effect of the goto statement during the activation of P within the for statement is undefined:

```
'BEGIN''PROCEDURE'P.,'BEGIN'... 'GOTO'L...'END'.,
        'FOR'I.=1'STEP'1'UNTIL'N'DO'
              L..'BEGIN'... P ...'END'
'END'
```

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). An actual parameter part may be empty or consist of up to 15 actual parameters. Where the procedure body is a statement written in ALGOL the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

4.7.3.1. Value Assignment (Call by Value)

All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and Types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these quasi-assignments (cf. following Note) were made to variables, arrays or labels local to the fictitious block with types as given in the corresponding specifications (cf. section 5.4.5.). As a consequence, variables, arrays or labels called by value are to be considered as nonlocal to the body of the procedure, but local to the fictitious block (cf. section 5. 4. 3.).

Note: If the formal parameter called by value is specified as a variable, the assignment is done as described in section 4. 2. If it is specified as an array, the assignment is done for each subscripted variable of that array as described in section 4. 2. (cf. 4.7.5.3). If it is specified as a label, the formal parameter is replaced throughout the procedure body by the label resulting as the value of the actual parameter (cf. 3.5.). Possible conflicts between the identifier of this label and identifiers occurring within the procedure body are handled as described in section 4. 7.3.2.

4.7.3.2. Name Replacement (Call by Name)

Any formal parameter not quoted in the value part is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this 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.

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. If the procedure is called from a place outside the scope of any nonlocal quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

Note: The following five examples demonstrate the application of the rules given in 4.7.3.1. to 4.7.3.3.

```
'BEGIN''COMMENT'EXAMPLE1.,
    'REAL'A, B.,
    'PROCEDURE'P(X, Y, Z).,'VALUE'Y., 'REAL'Z, Y, X.,
        'BEGIN'Z.=X+Y., Y.=X+Z., Z.=X+Y'END'.,
    A.=B.=1.,
    P(A+B, A, B).,
    OUTREAL(1, A)., OUTREAL(1, B).,
    P(A+B, A+B, A).,
    OUTREAL(1, A)., OUTREAL(1, B)
'END'
```

This example contains the declarations of a procedure P with three formal parameters X, Y, Z specified to correspond to 'REAL' type actual parameters, the second actual parameter (corresponding to Y) to be called by value. The body of this procedure declaration is the compound statement

```
'BEGIN'Z.=X+Y., Y.=X+Y., Z.=X+Y'END'
```

This procedure is activated twice by the procedure statements

P(A+B, A, B) and P(A+B, A+B, A).

The first of these statements results in the execution of a block consisting of (1) an assignment statement assigning the value of the second actual parameter A to a local variable Y and (2) the procedure body of P in which X is replaced by the first actual parameter A+B and Z by the third actual parameter B. The second procedure statement results in a similar block. So, this program is executed as:

```
'BEGIN''REAL'A, B.,
    A.=B.=1.,
    'BEGIN''REAL'Y., Y.=A.,
        'BEGIN'B.=(A+B)+Y., Y.=(A+B)+Y., B.=(A+B)+Y'END'
    'END.,
    OUTREAL(1, A)., OUTREAL(1, B).,
    'BEGIN''REAL'Y., Y.=A+B.,
        BEGIN'A.=(A+B)+Y., Y.=(A+B)+ Y., A.=(A+B)+Y'END'
    'END'.,
    OUTREAL(1, A)., OUTREAL(1, B)
'END'
```

It writes (cf. 6.2.2.2) after the first execution of the procedure body the values A=1, B=9 and after the second execution of the procedure body the values A=68, B=9. If the second parameter Y of the procedure P had not been specified to be called by value, then the first activation of P would yield other results (as the following example demonstrates), while the second activation would be impossible, as Y appears as a left part variable within the procedure body (cf. 4.7.5.2.) and the actual parameter A+B is an expression (not a variable).

```
'BEGIN''COMMENT'EXAMPLE2.,
    'REAL'A, B.,
    'PROCEDURE'P(X, Y, Z).,'REAL'X, Y, Z.,
        'BEGIN'Z.=X+Y., Y.=X+Y., Z.=X+Y'END'.,
     A.=B.=1.,
     P(A+B, A, B).,
     OUTREAL(1, A)., OUTREAL(1, B)
'END'
```

This program is executed as

```
'BEGIN''REAL'A, B.,
    A.=B.=1.,
    'BEGIN'B.=(A+B)+A., A.=(A+B)+A., B.=(A+B)+A'END'.,
    OUTREAL(1, A)., OUTREAL(1, B)
'END'
```

and writes the values A=5, B=13.

The following two examples demonstrate the meaning of "suitable systematic changes" in section 4.7.3.2. and 4.7.3.3.

```
'BEGIN''COMMENT'EXAMPLE3.,
'REAL'A.,
'PROCEDURE'B(C).,'REAL'C.,
      'BEGIN''REAL'A.,
            A.=1.,
            C.=A+C'END'.,
A.=2.,
  B(A)
'END'
```

This program is executed as

```
'BEGIN'
'R EAL' A.,
A.=2.,
'BEGIN''REAL' ACHANGED.,
 ACHANGED.=1.,
  A.=ACHANGED+A'END'
'END'
```

In the following example a procedure is called within an inner block:

```
'BEGIN''COMMENT'EXAMPLE 4.,
'REAL'A.,
'PROCEDURE'G(B)., REAL B.,
            B.=A.,

A.=1.,
   'BEGIN''REAL'A, C.,
    A.=2.,
    G(C)
    'END'
'END'
```

This program is executed as

```
'BEGIN''REAL' A.,
A.=1.,
'BEGIN''REAL'ACHANGE D, C.,
ACHANGED.=2
C.=A
'END'
'END'
```

The last example demonstrates the different actions of a formal parameter specified as a label if called by name or by value.

```
'BEGIN''COMMENT'EXAMPLE5.,
        'INTEGER'N.,
        'SWITCH'S.=S1, S2, S3, S4, S5.,
        'PROCEDURE'P(L1, L2).,'VALUE'L1.,'LABEL'L1, L2.,
             'BEGIN'N.=N+1.,
                  'IF'N'GREATER'5'THEN''GOTO'L1
                                       'ELSE''GOTO'L2
             'END'.,
        N.=1.,
        P(S(/N/), S(/N/)).,
        .
        .
        .
'END'
```

This program is executed as:

```
'BEGIN''INTEGER'N.,
        'SWITCH'S.=S1, S2, S3, S4, S5.,
         N.=1.,
        'BEGIN'N.=N+1.,
              'IF'N'GREATER'5'THEN''GOTO'S1
                                  'ELSE''GOTO'S(/N/).,
              'END'.,
              .
              .
              .
'END'
```

4.7.4. Actual-Formal Correspondence

The correspondence between the actual parameters of the procedure statement and the formal parameters 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.

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 ALGOL statement.

This imposes the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type specified for the corresponding formal parameter if called by value (cf. 5.4.5). Additionally, the kind and type of each actual parameter must be the same as that specified for the corresponding formal parameter, if called by name. This correspondence of kind and type of actual and formal parameters may be illustrated by the following table:

Specification of Formal Parameter	Allowed Actual Parameter	
	If Called by Name	If Called by Value
'STRING'	string	not allowed
'REAL'	arithmetic expression of type 'REAL'	any arithmetic expression
'INTEGER'	arithmetic expression of type 'INTEGER'	any arithmetic expression
'BOOLEAN'	Boolean expression	Boolean expression
'ARRAY' or 'REAL''ARRAY'	array identifier of type 'REAL'	array identifier of type 'REAL' or 'INTEGER'
'INTEGER''ARRAY'	array identifier of type 'INTEGER'	array identifier of type 'REAL' or 'INTEGER'
'BOOLEAN''ARRAY'	array identifier of type 'BOOLEAN'	array identifier of type 'BOOLEAN'
'LABEL'	designational expression	designational expression
'SWITCH'	switch identifier	not allowed
'PROCEDURE'	any procedure identifier	not allowed
'REAL''PROCEDURE'	function procedure identifier of type 'REAL'	function procedure identifier of type 'REAL' or 'INTEGER' (only a procedure with an empty parameter part)
'INTEGER''PROCEDURE'	function procedure identifier of type 'INTEGER'	same as for 'REAL PROCEDURE'
'BOOLEAN''PROCEDURE'	function procedure identifier of type 'BOOLEAN'	function procedure identifier of type 'BOOLEAN' (only a procedure with an empty parameter part)

Some important particular cases of the general rules are the following:

4.7.5.1. If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL 60 statement (as opposed to non-ALGOL code, cf. section 5.4.6.), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL-code, e. g. an input/output procedure (cf. section 6).

4.7.5.2 A formal parameter 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 parameter 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 (cf. 5.2.3.2.). In addition if the formal parameter is called by value the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4. A formal parameter which is used within the procedure body as a procedure identifier can only correspond to an actual parameter which is a procedure identifier of a procedure which has the same number of parameters, each of which is of the same type and kind.

4.7.5.5. A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier or a string, because these latter do not possess values (cf. 2.8.).

Note: The exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (c f. 5.4.1.) and which defines the value of a function designator (cf. 5.4.4.). This procedure identifier is in itself a complete expression.

Example.

```
'BEGIN''REAL''PROCEDURE'PI.,
        'COMMENT'THIS PROCEDURE CALCULATES THE VALUE
                    OF THE CIRCLE CONSTANT PI.,
        <procedure body>.,
  ' PROCEDURE'POWER(X) EXPONENT..(N) RESULT..(Y).,
        'VALUE'X, N.,
        'REAL'Y.,'INTEGER'N.,'REAL''PROCEDURE'X.,
        Y.=X'POWER'N.,
  'REAL'A, N.,
  'FOR'N.=0'STEP'0.5'UNTIL'5'DO'
        'BEGIN'POWER(PI, N, A)., OUTREAL(1, A)'END'
'END'
```

This example demonstrates that it is possible to call a formal parameter corresponding to a procedure identifier by value. The effect, however. would be the same if the formal parameter were specified 'REAL' instead of 'REAL''PROCEDURE'.

4.7.6. 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 and is handled like a comment (cf. 2.3).

5. DECLARATIONS

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. section 4. 1. 3.).

Dynamically this implies the following: at the time of an 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 goto statement) all identifiers which are declared for the block lose their local significance and retain the significance they had before entering the block.

Apart from labels (cf. 4.1.3.) and formal parameters of procedure declarations (cf. 5.4.5.) and with the exception of those for standard functions (cf. 3.2.4.) and input/output procedures (cf..1.2.), all identifiers of a program must be declared. No identifier may be declared more than once in any one block head, and no identifier occurring as a label within a block may be declared in the head of that block.

Notes: The order of the declarations within a block head is arbitrary.

Example. The block heads

'BEGIN''REAL'A.,'PROCEDURE'P(X).,'BOOLEAN'X.,<procedure body>.,
'INTEGER'I.,'PROCEDURE'PI.,<procedure body>.,

and 'BEGIN''INTEGER'I.,'REAL'A.,'PROCEDURE'PI.,<procedure body>.,
'PROCEDURE' P(X).,'BOOLEAN'X.,<procedure body>.,

are equivalent, even if, for example, the variable I occurs within the procedure body of P(X).

The value of a variable or an array is lost after exit from the block in which it is declared. If the block will be entered anew, the variable or array has no value until it is assigned a value again. If, on the other hand, the identifier of a variable or array gets a new significance on entering an inner block, the variable or array does not lose its value in the outer block, but is only inaccessible for the time being; it may be used with its old value after leaving the inner block.

Example.

```
'BEGIN''BOOLEAN'B.,
    B.='TRUE'.,
K..'BEGIN''INTEGER'M.,
        L..'IF'B'THEN''BEGIN'M.=1., B.='FALSE''END'
              'ELSE'M.=M+1.,
           'BEGIN''INTEGER'M.,
                  :
                  :
           'END'
            :
            :
           'GOTO'L.,
            :
            :
           B.='TRUE'.,
           'GOTO'K.,
             :
             :
        'END'
         :
         :
'END'
```

In this example M becomes a variable of type 'INTEGER', when entering the block labelled by the label K. This variable is then assigned the value 1. Since in the innermost block a new variable is declared by the identifier M, the former variable M is inaccessible during execution of that block. But on leaving the innermost block, M gets back the significance of the orignial variable with the value 1. When executing the statement 'GOTO'L, the block labelled K is not left and therefore M retains its value and the assignment M.=M+1 of the conditional statement may be executed. When, on the other hand, executing the statement 'GOTO'K, though the same sequence of statements will be executed, the block labelled K is left temporarily and therefore the variable M loses its value until the new assignment M.=1 is made.

Syntax

<declaration> ::= <type declaration> ! <array declaration> !
<switch declaration> ! <procedure declaration>

5.1. Type Declarations

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 identifiers to represent simple variables of a given type. Real declared variables may only assume positive and 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', (cf. also section 2.8.).

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable. Cf., however, the restrictions on actual parameters corresponding to formal parameters called by name (cf. 4.7.5.).

Examples. The examples of section 5.1.2. when used in the head of a block declare the identifiers P, Q, S to denote variables of type 'INTEGER' and the identifiers ACRYL, N to denote variables of type 'BOOLEAN' throughout the block.

Note: The block heads

'BEGIN''REAL'A., 'INTEGER'B., 'INTEGER'C., 'REAL'D

and 'BEGIN''REAL'A, D., 'INTEGER'C, B

are equivalent, i. e. the declarations of variables of the same type may be arbitrarily grouped together.

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> !
<type> 'ARRAY' <array list>

5.2.2. Examples

```
'ARRAY'A, B, C(/7..N, 2..M/), S(/-2..10/)
'INTEGER''ARRAY'A(/'IF'C'LESS'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 (cf. 3.1.4.) 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. An array may have up to 16 dimensions.

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.

5.3.4. Lower and Upper Bound Expressions

5.2.4.1. The expressions will be evaluated in the same way as subscript expressions (c f. section 3.1.4.2.).

5.2.4.2. The expression can only depend on variables and procedures which are nonlocal to the block for which the array declaration is valid because local variables do not have values before entering the statements of the block, and for example, the activation of a local procedure could make use of subscripted variables of the array before it is declared. Consequently, in the outermost block of a program only array declarations with constant bounds may be declared.

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

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

Examples. The first example of section 5.2.2. declares the identifiers A, B, C, to denote three two-dimensional arrays of type 'REAL' with identical subscript bounds and the identifier S to denote a one-dimensional array, also of type 'REAL', whose subscript may have values ranging from -2 to 10. The subscript bounds of A, B, C depend on the actual values of N and M when entering the block in the heading of which the declaration occurs. The values of the first subscript may range from 7 to N, the values of the second subscript from 2 to M. The subscript bounds do not change if N or M is assigned a new value during the execution of the block. But they will be evaluated anew if the block is left and entered again (cf. the notes in section 5.).

The second example of section 5.2.2. declares the identifier A to denote a one-dimensional array of type 'INTEGER'. The subscript values range either from 2 to 20 or from 1 to 20 depending on the sign of the value of C when entering the block in the heading of which the declaration occurs.

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'Q.=P, W
'SWITCH'S.=S1, S2, Q(/M/), 'IF'V'GREATER'-5'THEN'S3'ELSE'S4

5.3.3. Semantics

A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1, 2, ..., obtained by counting the items in the list from left to right. The value of the switch designator 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. A switch list may not consist of more than 16 designational expressions.

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.

Examples. The first example of section 5.3.2. defines the value of the switch designator Q(/1/) to be the label P and the value of Q(/2/) the label W. The value of, e.g., Q(/3/) or Q(/-5/) is undefined. The second example defines the values of the switch designators S(/1/), S(/2/) to be the labels S1, S2; the value of S(/3/) to be the value of Q(/M/), i.e., one of the labels P or W depending on the value of M at the time S(/3/) is referred to in any statement; and, finally, the value of S(/4/) to be one of the labels S3 or S4 depending on the value of V at the time S(/4/) is referred to. Thus, in the following sequence of statements:

```
          :
          :
      N.=0.,M.=5.,V.=3.,
          :
          :
S1..      ...
S2..      ...
P ..N.=N+2.,
          :
          :
W..M.=M-1.,
          :
          :
S3..V.=V-1.,
          :
          :
    'GOTO'S(/N/2/).,
          :
          :
S4..      :...
```

(assuming that they are preceded by the two switch declarations described in section 5.3.2. and that the variables N, M, V do not change their values by other statements than those explicitly stated), the statement 'GOTO'S(/N/2/) refers sequentially to the labels S1, S2, W, P, S3, S3, S3, S4.

5.3.5. Influence of Scopes

If a switch designator occurs outside the scope of a quantity entering into a designational expression in the switch list, and an evaluation of this switch designator selects this designational expression, then the conflicts between the identifiers whose declarations are valid at the place of the switch designator will be avoided through suitable systematic changes of the latter identifiers.

Example. The program

```
'BEGIN''SWITCH'A.=L.,
          :
          :
    L..'BEGIN''INTEGER'B.,
                :
                :
            L..B.=1.,
                'GOTO'A(/B/)
        'END'
          :
          :
'END'
```

is equivalent to the program

```
'BEGIN''SWITCH'A.=L.,
          :
          :
    L..'BEGIN''INTEGER'B.,
                :
                :
            LCHANGED..B.=1.,
                'GOTO'L
        'END'
          :
          :
'END'
```

5.4. Procedure Declarations

5.4.1. Syntax

<formal parameter> ::= <identifier>
<formal parameter list> ::= <formal parameter> !
    <formal parameter list> <parameter delimiter> <formal 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> ., !
    <specification part> ::= <specifier> <identifier list> .,
<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>

5.4.2. Example (see also the examples in APPRENDIX 3)

```
'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.=I+1'STEP'1'UNTIL'N'DO'
    'BEGIN'W.=A(/I,K/).,
            A(/I,K/).=A(/K,I/).,
            A(/K,I/).=W
    'END'
'END'TRANSPOSE

'INTEGER''PROCEDURE'STEP(U).,'REAL'U.,
    STEP.='IF'0'NOTGREATER'U'AND'U'NOTGREATER'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.,I.=K.=1.,
'FOR'P.=1'STEP'1'UNTIL'N'DO'
'FOR'Q.=1'STEP'1'UNTIL'M'DO'
'IF'ABS(A(/P,Q/))'GREATER'Y'THEN'
     'BEGIN'Y.=ABS(A(/P,Q/).,I.=P.,K.=K.=Q'END'
'END'ABSMAX

'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'INNER PRODUCT
```

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 logically represented by the delimiter 'CODE' (cf. 5.4.6.), the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occuring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (c f. section 3.2. Function Designators and section 4.7. Procedure Statements) be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal parameters will be either local or nonlocal to the body depending on whether they are declared within the body or not. Those of them which are nonlocal to the body may well be local to the block in the head of which the procedure declaration appears (however, they may not be local to the block or procedure body nested within that block, even if the procedure is activated only in the nested block or procedure body). The procedure body always acts like a block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3.), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

Note: According to section 2.4.3. all formal parameters of a procedure declaration have to be distinct, i.e., no identifier may occur twice in a formal parameter part. In the case of a function procedure (cf. 5.4.4.), the procedure identifier also has to be different from all formal parameters.
A procedure may be declared without any formal parameters.

Example. If division by the same variable, called DENOMINATOR, occurs frequently in a program, it might be useful to precede each statement in which such division is executed by the following procedure (cf.6.2.2.5.):

```
'PROCEDURE'ERRORPRINT.,
   'IF'DENOMINATOR'EQUAL'0'THEN'
      'BEGIN'OUTSTRING(1,'('ZERODIVIDE')')
            'GOTO'END
'END'
```

Cf. also the example of section 4.7.5.4.

5.4.3.1. Recursive Procedures

As in a procedure body, identifiers may be used which are local to the block in the head of which the procedure declaration occurs, especially a procedure may activate itself within its procedure body. In this case, when activating the procedure the actions described in sections 4.7.3.1. to 4.7.3.3. are performed repeatedly. If the procedure body is a block, this results in forming a series of nested blocks. A similar situation exists if, for example, two procedures activate each other within their procedure bodies.

Example.

```
'PROCEDURE'P(X, Y).,
    'VALUE'X.,'INTEGER'X, Y.,
    'BEGIN''INTEGER'N.,
       'IF'X'EQUAL'1'THEN'Y.=1
          'ELSE''BEGIN'P(X-1, N).,
                        Y.=X*N
                'END'
'END'
```

This procedure assigns the factorial of the first parameter to the second parameter (though it is not a good program for this problem). The procedure statement

```
P(3, F)
```

will result in the following execution of three nested copies of the procedure body of P:

```
'BEGIN''INTEGER'X1.,X1.=3.,
'BEGIN''INTEGER'N1.,
   'IF'X1'EQUAL'1'THEN'F.=1'ELSE'
      'BEGIN''BEGIN''INTEGER'X2.,X2.=X1-1.,
            'BEGIN''INTEGER'N2.,
               'IF'X2'EQUAL'1'THEN'N1.=1'ELSE'
                'BEGIN''BEGIN''INTEGER'X3.,X3.=X2-1.,
                        'BEGIN''INTEGER'N3.,
                        'IF'X3'EQUAL'1'THEN'N2.=1'ELSE'
                        'BEGIN'P(X3-1, N3).,
                         N2.=X3*N3'END'
                       'END''END'P(X2-1, N2).,
                         N1.=X2*N2
                'END'
      'END''END'P(X1-1, N1).,
      F.=X1*N1
  'END'
'END''END'P(3, F)
```

Note: If a program is meaningful, the recursive activation of a procedure within its body has to be executed in a conditional form, otherwise, there would be an infinite number of activations performed. The body replacement as described in section 4.7.3.3. has to be performed for each recursive step only if the procedure is really activated in that step.

5.4.4. Values of Function Designators

For a procedure declaration to define the value of a function designator (cf. 3.2.) there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in a left part; at least one of these must be executed, and the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

Notes: A procedure declared to define the value of a function designator may be activated both in defining a function designator occurring in an expression or in the form of a procedure statement. In the latter case the value assigned to the procedure identifier is lost.

Examples. The third example of section 5.4.2. is a procedure declaration defining a function designator. If E is an arithmetic expression of type 'REAL' then the value of STEP(E) is 1 if $0 \leq E \leq 1$ and 0 if $E<0$ or $E>1$.

The following example is incorrect:

```
'INTEGER''PROCEDURE'FACTORIAL(N).,
'VALUE'N.,'INTEGER'N.,
    'BEGIN''INTEGER'I., FACTORIAL.=1.,
      'FOR'I.=1'STEP'1'UNTIL'N'DO'
       FACTORIAL.=FACTORIAL*I
    'END'
```

In the procedure body of this declaration the procedure identifier occurs in another position than in a left part list. It cannot be activated in that position, since the actual parameter part is missing. A correct form of that procedure declaration is (cf. also

the example in the note of section 5.4.3.):

```
'INTEGER''PROCEDURE'FACTORIAL(N).,
  'VALUE'N.,'INTEGER'N.,
  'BEGIN''INTEGER'I, F.,
FACTORIAL.=F.=1.,
     'FOR'I.=1'STEP'1'UNTIL'N'DO'
  FACTORIAL.=F.=F*I
  'END'
```

5.4.4.1 Side effects

Within the body of a procedure declaration, an assignment of a value to a variable, which is neither local to the procedure body nor a formal parameter of that procedure to be called by value, or a goto statement referring finally to a label, which is not local to the procedure body, is called a "side effect" of the procedure. Side effects of a procedure are generally undefined in the ALGOL language if this procedure is used to define the value of a function designator.

Notes: This rule applies to "implicit" side effects, too, i. e., such ones which are caused within the procedure body by activation of another procedure, e.g. an input/output procedure.

The rule does not apply if a procedure is activated in the form of a procedure statement, even if it is declared to define the value of a function designator.

5.4.5. Specifications

In the heading a specification part, giving information about the kinds and types of all formal parameters by means of an obvious notation, is to be included, if the formal parameter part is not empty. In this part no formal parameter may occur more than once. Within the procedure body a formal parameter specified in the heading of the procedure declaration may occur in any syntactic context, where an identifier declared by the corresponding declaration might occur.

Notes: Here, the specification 'LABEL' corresponds, of course, to the implicit declaration of labels (cf.4.1.3.). The specification 'STRING' does not correspond to any declaration of an identifier; regarding the occurrence of formal parameters specified 'STRING' cf. section 4.7.5.1. Cf., also, the correspondence tables for formal and actual parameters in section 4.7.5.

5.4.6. Precompiled Procedures

It is possible to translate a procedure declaration by a separate process of the System/360 Operating System ALGOL Compiler. Such a "precompiled" ALGOL procedure may be used by one or more ALGOL programs. In this case, within these programs the corresponding procedure declaration consists of the declarator 'PROCEDURE' or <type>'PROCEDURE', the normal procedure heading, and the delimiter 'CODE', which represents the precompiled procedure body.

Parameters of such procedures may not be specified 'PROCEDURE', <type> 'PROCEDURE', or 'SWITCH'. No different precompiled procedures used by an ALGOL program may be denoted by the same identifier, even if their scopes are disjoint.

INPUT/OUTPUT PROCEDURES

The transmission of data to and from an external medium that is not directly accessible by the ALGOL program is achieved by calls to input/output procedures.

6.1. General Characteristics

6.1.1. Characters, Data Sets, Records

6.1.1.1. For each external medium there exists a representation of a set of at most 256 characters. Within this character set there are available at least the 26 letters, 10 digits, a blank space and those special characters, which constitute the basic symbols, e.g., + and '. Further characters corresponding to symbols allowed in strings may be available (cf. strings 2.6.3.). The characters compose data on the external medium according to the formats in the sections describing the different input/output procedures (cf. 6.2.2.).

6.1.1.2. Externally, the data to be transmitted is combined into "data sets". A data set is a named collection of logically related data which to the user appears to be in a continuous string. The external organization of the data is maintained automatically and need not concern the user of the ALGOL Language. Within an ALGOL program each data set is uniquely identified by an integer, the "data set number" [4] . Only the numbers 0 to 15 are allowed. [5].

---

4) Each "data set number" of this publication corresponds uniquely to a "data definition name" of the Job Control Language (cf. IBM Operating System/360 Concepts and Facilities, Form C28-6536). Although one data definition name may represent different data sets (for different executions of a program, or even for one execution of a program in the case of concatenated data sets), within a single execution of a single ALGOL program the correspondence between a data definiton name and data set can be considered unique. Therefore, in this publication the terms "data set" and "data set number" are used instead of the term "data definiton name". Also the phrase "opening a data set" instead of "opening a data control block" is used unambiguously (c f. Concepts and Facilities) .

5) The data set number 0 is reserved for the system input data set (with data definition name SYSIN, cf. IBM Operating System/360, Job Control Language. Form C28-6539). Only sequential input from this data set is possible; any output or backwards repositioning requests specifying data set number 0 are undefined.
The data set number 1 is reserved for the system output data set (with data definition name SYSPRINT), which will contain object program diagnostic messages. It is assumed that this data set will be printed. Only sequential output to this data set is possible; any input or backwards repositioning requests specifying data set number 1 are undefined.

6.1.1.3. In general, the continuous string data composing a data set is logically split into records. A record contains the information processed as a unit by the program. The intended use of a record often influences its length; there are of course certain relations to the characteristics of the external device of the data set (details will be stated in a later publication). For example, an 80 column card image may be a record. If a record may be printed, it should be restricted in length to a print line. If a record is used simply to transmit data from one program to another, its length is governed by the maximum number of items of data it may contain. The mechanics of transferring records to and from the external medium is maintained automatically and needs not concern the user of the ALGOL language. The number of characters within a record is fixed for all records of data set; this number is called the record length P; it may in no case exceed 32.760. The programmer may specify the value of P (within the physical limits of the external medium) by executing the standard procedure SYSACT with FUNCTION=6 (cf. 6.3.2.6.) before creating a data set. Otherwise, a standard record length of 80 characters is chosen when the data set is created by the first transfer of data to it. If the data set was created by another program, the value of P was specified by the creating program and is valid for the current ALGOL program. [6)]

6.1.1.4. Additionally, the programmer can split a data set into "sections" by specifying a fixed number Q of records to be in one section. This is done by executing the standard procedure SYSACT with FUNCTION=8(cf. 6.3.2.8.) before creating the data set. This specification is most meaningful if the data set is intended to be printed; the contents of one section normally will appear on one page. Any repositioning backwards within the current ALGOL program is undefined for a data set split into sections. Therefore (according to section 6.2.1.3.) only output procedures may process such data sets.

---

6) The specification of the record length by the ALGOL program is not possible if it is determined by the data set label of an already existing data set or by the corresponding data definiton control statement (cf. IBM Operating System/360 Concepts and Facilities, Form C28-6535).

6.1.1.5. Each data set is in one of three states, "open", "close" or "exhausted. As long as a data set is open, it is logically connected with the ALGOL program, i.e. data can be transferred to or from it. As long as a data set is closed, it is logically disconnected, i. e. no transfer of data can be made. As long as a data set is exhausted, it is logically connected with the ALGOL program as explained above, but there are no more data available for input from the data set to the ALGOL program; then any input procedure statement is undefined.

A data set becomes open either by a call to the procedure SYSACT with FUNCTION=12 (cf.6.3.2.12) or, if it is not yet open, automatically by execution of any procedure requesting transfer of data with that data set.

A data set becomes closed either by a call to the procedure SYSACT with FUNCTION=12 (cf. 6.3.2.12) or, if it is still open, automatically at the end of the ALGOL program.

A data set that has been closed can be opened again either within the current ALGOL program or within a later program.

A data set becomes exhausted by execution of an input procedure transferring the last data in the data set to the ALGOL program. It becomes normally open again by any repositioning backwards.

6.1.2. Standard Procedure Identifiers

Certain identifiers are used for standard procedures. These procedures are available without explicit declaration. They are considered as declared in a block surrounding the whole program. If the identifier is declared in any block head as something different (e. g. an array) the standard function it represents is unavailable throughout that block.

There are four classes of standard procedures:

input procedures:	INSYMBOL, INREAL, ININTEGER, INBOOLEAN, INARRAY, INTARRAY, INBARRAY
output procedures:	OUTSYMBOL, OUTREAL, OUTINTEGER, OUTBOOLEAN, OUTSTRING, OUTARRAY, OUTTARRAY, OUTBARRAY
control procedure:	SYSACT
procedures for intermediate storage:	PUT, GET

6.2. Input Procedures and Output Procedures

6.2.1. General Characteristics

6.2.1.1. The input and output procedures transfer data from or to a data set, which is specified by the data set number used as the first actual parameter in the procedure statement. The data set is handled by these procedures sequentially as explained in the following paragraph:

For each data set that is open (cf. 6.1.1.4.), a record pointer S and a character pointer R are maintained. When the data set is opened, the record pointer points to the first record of the data set (S=1), and the character pointer to the first character within that record (R=1). Each time a character is transmitted to or from the data set, R is increased by 1, until it equals the record length P. When R=P and a chracter is transmitted, S is increased by 1 and R is reset to 1.

Note: The transfer of data to or from the external device is related to any change of S; i. e. any record is actually transferred as a whole at this time.

These two pointers uniquely denote the location of a character within the data set. Each input or output procedure transfers a sequence of one or more characters, starting at the location denoted by the current position of the two pointers.

This sequential order of handling a data set can be changed by altering one of the two pointers R and S through the execution of the control procedure SYSACT (cf. 6.3.).

If a data set has been split into sections of Q records each (cf. 6.1.1.4), the record pointer S is set back to 1 if a section has been filled, i. e., S counts the records within the sections only and never exceeds the specified value of Q. Since in this case the value of S does not uniquely denote the record within a data set (only within a section), altering of the record pointer by SYSACT with FUNCTION=4 is undefined and no repositioning backwards is possible (cf., however, 6.3.2.14 and 6.3.2.15).

6.2.1.2. If a data set is suitable both for input and output (e.g. on tape), all data transferred to that data set by an output procedure can be transferred back by an input procedure, provided that an appropriate repositioning of the data set has been performed by execution of the control procedure SYSACT (cf. 6.3.). If in this case, output and input are performed by a pair of corresponding procedures, the value of the data transferred remain unchanged throughout the process (cf. 2.8. for 'REAL' quantities).

Corresponding input and output procedures are:

OUTSYMBOL	-	INSYMBOL
OUTREAL	-	INREAL
OUTINTEGER	-	ININTEGER
OUTBOOLEAN	-	INBOOLEAN

6.2.1.3. Each transfer of data to a data set by means of an output procedure (after repositioning the data set by SYSACT, cf. 6.3.) destroys all data in the data set located beyond the data just transferred, i. e. the data, whose location is represented by values Sl, Rl of the record and character pointer for which the relations Sl > S or Sl = S, Rl > R hold, where S and R are the current values of the two pointers.

6.2.1.4. For several input and output procedures, described in the following sections, a sequence of K or more blank spaces contained in the data set serves as a delimiter of the data to be transferred, while less than K blank spaces are ignored. In this connection, for each data set, K is a positive integer assumed to be two unless specified differently by execution of the procedure SYSACT with FUNCTION= 10 (cf. 6.3.2.10). The end of a record also serves as delimiter of the data to be transferred. (For details, cf. the sections 6.2.2.2., 6.2.2.3., 6.2.2.4. describing the individual procedures).

6.2.2. Description of the Input Procedures and Output Procedures.

6.2.2.1. Procedures INSYMBOL and OUTSYMBOL

6.2.2.1.1. Assumed Procedure Declarations

```
'PROCEDURE' INSYMBOL (DATA SET NUMBER, STRING,
                      DESTINATION).,
  'VALUE' DATA SET NUMBER.,
  'INTEGER' DATA SET NUMBER, DESTINATION.,
  'STRING' STRING.,
  <procedure body>.,
```

```
'PROCEDURE' OUTSYMBOL (DATA SET NUMBER, STRING, SOURCE).,
   'VALUE' DATA SET NUMBER, SOURCE.,
   'INTEGER' DATA SET NUMBER, SOURCE.,
   'STRING' STRING.,
   <procedure body>.,
```

6.2.2.1.2. Semantics

The two procedures INSYMBOL and OUTSYMBOL provide the means of communication between an external data set and a variable of the program, or more generally an expression in case of OUTSYMBOL, in terms of single symbols. The first actual parameter of either procedure specifies the data set by its data set number; the second actual parameter is a string denoting the conversion between the symbols of the external medium and the program as explained below; the third actual parameter is the variable or expression. In either procedure the correspondence between the characters of the specified data set and the values of the expression is established by mapping one-to-one the sequence of symbols of the string between the outermost string quotes, taken from left to right, into the sequence of positive integers 1, 2, 3, ... Using this correspondence, the procedure INSYMBOL assigns to the third actual parameter the value corresponding to the current character within the specified data set (denoted by the current position of the record and character pointers). If the symbol corresponding to this character does not appear in the string given as the second actual parameter, the value zero is assigned. Similarly the procedure OUTSYMBOL transfers the symbol corresponding to the value of the third actual parameter to the current character position within the specified data set (denoted by the record and character pointers). In this case the value zero corresponds to the blank space; negative values or values greater than the length of the specified string yield undefined results. Finally, either procedure sets the record and character pointer to the next character position within the data set.

6.2.2.1.3. Example.

The statement:

```
'FOR'I. = 1'STEP'1'UNTIL'15'DO'
'BEGIN' INSYMBOL (0, '('ABCDEFGHIJKL')', V).,
OUTSYMBOL (1,'('1234567890+')', V)'END'.,
```

transfers the following character sequence, which appears on the data set with the data set number 0:

... AXBIDA+4EFMJ5FK...

to the following character sequence on the data set with the data set number 1:

...1b2941bb56b0b6+...

(where b denotes a blank space).

6.2.2.2. Procedures INREAL and OUTREAL

6.2.2.2.1. Assumed Procedure Declarations

```
'PROCEDURE' INREAL (DATASETNUMBER, DESTINATION).,
   'VALUE' DATASETNUMBER.,
   'INTEGER' DATASETNUMBER.,
   'REAL' DESTINATION.,
   <procedure body>.,

'PROCEDURE' OUTREAL (DATASETNUMBER, SOURCE).,
   'VALUE' DATASETNUMBER, SOURCE.,
   'INTEGER' DATASETNUMBER.,
   'REAL' SOURCE.,
   <procedure body>.,
```

6.2.2.2.2. Semantics

The two procedures INREAL and OUTREAL transfer values of type 'REAL' between the external data set specified by the first actual parameter and the variable of the program, or more generally the expression in case of OUTREAL, appearing as the second actual parameter.

The procedure INREAL scans the specified data set sequentially,beginning at the current position of the record and character pointers, until it finds the first characters written according to the syntax of number (cf. 2.5.1.). This number can be as large as the syntax permits; the first character not allowed syntactically serves as a delimiter. In addition either of the the two delimiters described in section 6.2.1.4 (i.e., either K or more blank spaces or the record end) are considered to delimit the number. If no syntactically correct number is found in front of one of these delimiters, the scanning resumes with the characters after the delimiter. The number found by this process is converted to type 'REAL' and its value assigned to the second actual parameter. Then, the delimiter of the number in the data set is skipped so that the record and character pointers after execution of INREAL point to the character following the delimiter.

The procedure OUTREAL converts the value of the second actual parameter to a 'REAL' type number written in an external standard format. A number written

in this format occupies a field of 13 (short form) or 22 (long form) consecutive character positions [7] and consists of the sign, the first significant digit, the decimal point, either 6 or 15 following digits, the separator ',the sign and the two digits of the decimal exponent (scale factor), written in this order. If the value of the number is zero, the digit 0 is written in the second character position of this field and the rest of the field is filled by blank spaces. The number written in this format is transferred to the field starting at the current position of the record and character pointers within the specified data set. If there is not enough room in the current record, (i.e. if the difference between the record length P (c f.6.1.1.3.) and the current value of the character pointer R minus one, P - (R-1), is less than the field length of the standard format described), the remaining character positions within the current record are filled by blank spaces and the number to be transferred is written starting at the first position of the next record. Finally, if K character positions are left in the current record following the field of the number transferred they are filled by blank spaces, that serve as delimiter of the number (cf. 6.2.1.4). If less than K character positions are left, they are filled by blank spaces and the record end is the delimiter. Then the record and character pointers are set to the character position following the delimiter.

6.2.2.2.3. Example

The statement:

'FOR'I.=1'STEP'1'UNTIL'6'DO'
'BEGIN'INREAL(0,V)., OUTREAL(1,V)'END'.,
transfers the following character sequence, which appears on the data set with the data set number 0:
...1, -bb034.5b'5ABC+-7.'7A0b-'1+X...
to the following character sequence on the data set with the data set number 1 (assuming K = 2 for both data sets and the short form of 'REAL' type numbers is used:
...+1.000000'+00bb+3.450000'+06bb
-7.000000'+00bb+1.000000'+07bb
b0bbbbbbbbbbbbb+1.000000'+01bb...

In the input string the first number, 1, is delimited by the comma. The following minus sign is eliminated by two blank spaces so that the sign of the following number, 34.5'5, is plus; this number is interspersed by a non-significant blank

---

[7] The programmer may specify when invoking the System/360 Operating System ALGOL compiler if calculations involving 'REAL' type numbers shall be executed with a precision of 7 or 16 significant decimal digits.

space and delimited by the character A. The following number, -7, is delimited by the decimal point since syntactically a decimal point without following digits is not allowed (cf. 2.5.1.). The next number, '7, is delimited by the character A. The next number, 0, is followed by a non-significant blank space and delimited by the following minus sign. Since this delimiter - is skipped the sign of the last number, '1, is plus.

6.2.2.3. Procedures ININTEGER and OUTINTEGER

6.2.2.3.1. Assumed Procedure Declarations

```
'PROCEDURE'ININTEGER(DATASETNUMBER, DESTINATION).,
   'VALUE'DATASETNUMBER.,
   'INTEGER'DATASETNUMBER, DESTINATION.,
   <procedure body>.,

'PROCEDURE'OUTINTEGER(DATASETNUMBER, SOURCE).,
    'VALUE''DATASETNUMBER, SOURCE.,
    'INTEGER'DATASETNUMBER, SOURCE.,
    <procedure body>.,
```

6.2.2.3.2. Semantics

The two procedures ININTEGER and OUTINTEGER transfer values of type 'INTEGER' between the external data set specified by the first actual parameter and the variable of the program, or more generally the expression in case of OUTINTEGER, appearing as the second actual parameter.

The actions of the procedures ININTEGER and OUTINTEGER are exactly the same as those of the procedures INREAL and OUTREAL, respectively, (cf. 6.2.2.2.2.), except that the number found by the scanning process or the value to be transferred, respectively, is converted to type 'INTEGER' instead of 'REAL' and that the value written by OUTINTEGER is in another external standard format that occupies a field of 11 consecutive characters (instead of 13 or 22). It starts with a sign followed by at most 10 significant digits written into the last character positions of the field; the initial positions are filled with blank spaces if necessary. If the value of the number is zero, the digit 0 is written in the last character position of this field, and the leading nine positions are filled with blank spaces.

6.2.2.3.3. Example

The statement:

```
'FOR'I.=1'STEP'1'UNTIL'5'DO'
'BEGIN'ININTEGER(0, V)., OUTINTEGER(1, V)'END'.,
```

transfers the following character sequence, which appears on the data set with the data set number 0:

...54bb-b3.7,.bb'9-65.4'4bb0,...
to the following character sequence on the data set with the data set number 1
(assuming K=2 for both data sets):
...bbbbbbbb+54bbbbbbbbbbb-4bb
+1000000000bbbbbb+654000bb
bbbbbbbbbb0bb...

6.2.2.4. Procedures INBOOLEAN and OUTBOOLEAN

6.2.2.4.1. Assumed Procedure Declarations

```
'PROCEDURE'INBOOLEAN(DATASETNUMBER, DESTINATION).,
   'VALUE' DATA SET NUMBER.,
   'INTEGER' DATA SET NUMBER.,
   'BOOLEAN' DESTINATION.,
   <procedure body>

'PROCEDURE' OUTBOOLEAN (DATA SET NUMBER, SOURCE).,
   'VALUE' DATA SET NUMBER, SOURCE.,
   'INTEGER' DATA SET NUMBER.,
   'BOOLEAN' SOURCE.,
   <procedure body>.,
```

6.2.2.4.2. Semantics

The two procedures INBOOLEAN and OUTBOOLEAN transfer values of type 'BOOLEAN' between the external data set specified by the first actual parameter and the variable of the program, or more generally the expression in case of OUTBOOLEAN, appearing as the second actual parameter.

The action of the procedure INBOOLEAN is the same as that of the procedure INREAL, except that it scans for characters written according to the syntax of logical value (cf. 2.2.2.) and the information so found is converted to type 'BOOLEAN' before assigning it to the second actual parameter.

The procedure OUTBOOLEAN converts the value of the second actual parameter to its representation written in an external standard format. A value written in this format occupies a field of seven consecutive characters instead of 13 or 22 and consists of the characters 'TRUE'b or 'FALSE'. The further action of OUTBOOLEAN is exactly the same as that of OUTREAL.
Note: Blank spaces are handled by the procedure INBOOLEAN and OUTBOOLEAN in the same way as by the procedures INREAL and OUTREAL: For INBOOLEAN, K more blank spaces serve as delimiter and cause a new start of scanning if no complete logical value had been found before. For OUTBOOLEAN, K blank spaces are transferred following the logical value, if there is enough space left in the current record.

6.2.2.4.3. Example.

The statement:

'FOR'I.=1'STEP' 1'UNTIL'3'DO'
'BEGIN'INBOOLEAN(0, V)., OUTBOOLEAN(1, V)'END'.,
transfers the following character sequence, which appears on the data set with the data set number 0:

...'TRUE'ABC'FbAbLbSbE','TRbbUE'FALSE'...

to the following sequence on the data set with the data set number 1 (assuming K=2 for both data sets):

...'TRUE'bbb'FALSE'bb'FALSE'bb...

6.2.2.5. Procedure OUTSTRING

6.2.2.5.1. Assumed Procedure Declaration

```
'PROCEDURE'OUTSTRING (DATA SET NUMBER, STRING).,
   'VALUE' DATA SET NUMBER.,
   'INTEGER' DATA SET NUMBER.,
   'STRING' STRING.,
   <procedure body>.,
```

6.2.2.5. Semantics

The procedure OUTSTRING transfers the symbols between the outermost string quotes of the string appearing as the second actual parameter to the data set specified by the first actual parameter, starting at the position currently denoted by the record and character pointers.

Each symbol of the string is transmitted to one character position. The string may span one or more consecutive records. Finally, the record and character pointers are set to the character position following the last character transmitted.

6.2.2.5.3. Example

The statements
```
OUTSTRING (1, '('ALGOLb')').,
OUTSTRING (1, '('REPORT')').,
```
transfer the following character sequence to the data set with the data set number 1:

...ALGOLbREPORT...

6.2.2.6. Procedures for Array Transmission

6.2.2.6.1. Assumed Procedure Declarations

```
'PROCEDURE' INARRAY (DATA SET NUMBER, DESTINATION).,
   'VALUE' DATA SET NUMBER.,
   'INTEGER' DATA SET NUMBER.,
   'ARRAY' DESTINATION.,
    <procedure body>.,

'PROCEDURE' OUTARRAY (DATA SET NUMBER, SOURCE).,
   'VALUE' DATA SET NUMBER, SOURCE.,
   'INTEGER' DATA SET NUMBER.,
   'ARRAY' SOURCE.,
    <procedure body>.,

'PROCEDURE' INTARRAY (DATA SET NUMBER, DESTINATION).,
   'VALUE' DATA SET NUMBER.,
   'INTEGER' DATA SET NUMBER.,
   'INTEGER' 'ARRAY' DESTINATION.,
    <procedure body>.,

'PROCEDURE'OUTTARRAY(DATA SET NUMBER, SOURCE).,
   'VALUE' DATA SET NUMBER, SOURCE.,
   'INTEGER' DATA SET NUMBER.,
   'INTEGER' 'ARRAY' SOURCE.,
   <procedure body>.,

'PROCEDURE' INBARRAY (DATA SET NUMBER, DESTINATION).,
   'VALUE' DATA SET NUMBER.,
   'INTEGER' DATA SET NUMBER.,
   'BOOLEAN' 'ARRAY' DESTINATION.,
    <procedure body>.,

'PROCEDURE' OUTBARRAY (DATA SET NUMBER, SOURCE).,
   'VALUE' DATA SET NUMBER, SOURCE.,
   'INTEGER' DATA SET NUMBER.,
   'BOOLEAN' 'ARRAY' SOURCE.,
   <procedure body>.,
```

6.2.2.6.2. Semantics

The procedures INARRAY, OUTARRAY, INTARRAY, OUTTARRAY, INBARRAY, OUTBARRAY transfer values between the external data set specified by the first actual parameter and the array of the program appearing as the second actual parameter.

Assuming the array ARRAY has been declared to have the lower bounds L1, L2, ..., LN, in this order, [8] and the upper bounds U1, U2, ..., UN, in this order, the procedure statement:

INARRAY (DATASETNUMBER, ARRAY).,

is equivalent to the block:

```
'BEGIN' 'INTEGER' I1, I2, ..., IN.,
    'FOR'I1.=L1'STEP'1'UNTIL'U1'DO'
    'FOR'I2.=L2'STEP'1'UNTIL'U2'DO'
    ...
    'FOR'IN.=LN'STEP'1'UNTIL'UN'DO'
      INREAL(DATASETNUMBER, ARRAY(/I1, I2, ..., IN/))
'END'.,
```

The same relation as between the procedures

INREAL and INARRAY

exists between the procedures:

OUTREAL and OUTARRAY,
ININTEGER and INTARRAY,
OUTINTEGER and OUTTARRAY,
INBOOLEAN and INBARRAY,
OUTBOOLEAN and OUTBARRAY.

---

8) For descriptive purpose three underscored points are used here, these points do not have the syntactical meaning of points, but are used as an ellipsis, i.e. "etc".

Note: The possibly multidimensional structure of the array within the ALGOL program is not reflected in the corresponding data in the external data set, where it appears only in a linear sequence.

6.2.2.6.3. Example:

The following program:

```
'BEGIN''INTEGER'I, N, R, S.,
   'INTEGER''ARRAY' A(/1..5/), B(/0..1, -1..+1/).,
   'FOR' I.=1'STEP'1'UNTIL'5'DO'
        A(/I/).=I.,
   N.=0.,
   SYSACT (2, 1, R)., SYSACT (2,13,S).,
   OUTTARRAY(2, A)., OUTINTEGER (2,N).,
   SYSACT (2,4, S)., SYSACT (2, 2, R).,
   INTARRAY (2, B)
'END'
```

transfers first the following sequence of characters to the data set with the data set number 2 (assuming K=2):

```
... bbbbbbbbb+1bbbbbbbbbbb+2bb
    bbbbbbbbb+3bbbbbbbbbbb+4bb
    bbbbbbbbb+5bbbbbbbbbbb+0bb...
```

and transfers secondly, after repositioning (c.f. SYSACT, 6.3.), the corresponding values back, assigning the following values to the array B:

B(/0, -1/).=1, B(/0, 0/).=2, B(/0,1/).=3,
B(/1,-1/).=4, B(/1, 0/).=5,B(/1,1/).=0.

6.3. Control Procedure SYSACT

In order to obtain finer control over the input/output processes, the control procedure SYSACT can be used to gain access to certain "system parameters", to influence the status of a data set and, especially, to alter the sequential order of data transfer with a data set. These system parameters, which are maintained

internally for each data set, are:[9)]

the character pointer R	(cf. 6.2.1.1)
the record pointer S	(cf. 6.2.1.1)
the record length P	(cf. 6.1.1.3)
the number of records per section Q	(cf. 6.1.1.4)
the number of blanks serving as delimiter K	(cf. 6.2.1.4)
the state (open or closed) of a data set C	(cf. 6.1.1.5)

(C =1 means "open", C=0 means "close"
C=-1 means "exhausted")

6.3.1. Assumed Procedure Declaration

'PROCEDURE' SYSACT (DATA SET NUMBER, FUNCTION, QUANTITY).,
'VALUE' DATA SET NUMBER, FUNCTION.,
'INTEGER' DATA SET NUMBER, FUNCTION, QUANTITY.,
<procedure body>

6.3.2. Semantics

The first parameter DATA SET NUMBER specifies the effected data set. The second parameter FUNCTION specifies the special action of a SYSACT procedure statement. The third parameter QUANTITY is a variable of the program, or in special cases more generally an expression, which sets or records the values of the system parameters R, S, P, Q, K, and C (cf. footnote 9).

The following table summarizes the actions of SYSACT depending on the value of FUNCTION. For detailed definitions, see sections 6.3.2.1. to 6.3.2.15.

---

9) The letters R, S, P, Q, K, C used for the system parameters are not identifiers in the syntactic sense of 2.4. and not variables in the syntactic sense of 3.1. They do not have a predescribed meaning in any ALGOL program, but serve only for descriptive purposes within this publication. In the following sections ALGOL-like statements containing these letters are used, such a statement as S. = QUANTITY., is not a valid ALGOL statement and does not mean an assignment to a declared variable S, but serves only to indicate that the record pointer is set to the position denoted by the value of quantity (and that the corresponding internal action, i. e. repositioning of the data set, is executed).

Value of FUNCTION	Action of SYSACT
1	QUANTITY.=R
2	R.=QUANTITY
3	QUANTITY.=S
4	S.=QUANTITY
5	QUANTITY.=P
6	P.=QUANTITY
7	QUANTITY.=Q
8	Q.=QUANTITY
9	QUANTITY.=K
10	K.=QUANTITY
11	QUANTITY.=C
12	C.=QUANTITY
13	QUANTITY.=S and internal action
14	skip records
15	skip to next section

6.3.2.1. Value of FUNCTION=1.
QUANTITY.=R.,

A SYSACT procedure statement with FUNCTION=1 is defined only if the specified data set is open and the third actual parameter QUANTITY is a variable. The procedure statement assigns the value denoted by the current position of the character pointer R to QUANTITY.

6.3.2.2. Value of FUNCTION=2.

A SYSACT procedure statement with FUNCTION=2 is defined only if the specified data set is open and QUANTITY has a positive 'INTEGER' type value not exceeding the record length P for the specified data set. The action of the statement depends on whether the last input/output procedure statement (not SYSACT) for the specified data set was input or output. (If input or output has not occurred since opening, i.e., since a SYSACT procedure statement with FUNCTION=12, QUANTITY=1 (cf. section 6.3.2.12), input is assumed.)

6.3.2.2.1. Action after Input.

'IF'QUANTITY'NOTGREATER'R'THEN'S.=S+1.,
R.=QUANTITY.,

The character pointer R is set to the position denoted by the value of QUANTITY ($1 \leq$ QUANTITY $\leq$ P). The record pointer S is left unchanged or increased by 1 depending on the relation between the old and new position of the character pointer R. The contents of the data set itself is not changed. A

following input or output procedure statement will start by handling that character denoted by character pointer position QUANTITY, which follows next to the old position of the data set.

6.3.2.2.2. Action after Output.

```
'FOR'I.=R'STEP'1'UNTIL'QUANTITY-1+
      ('IF' QUANTITY'GREATER'R'THEN'0'ELSE'1)'DO'
 OUTSYMBOL (DATA SET NUMBER,'('b')',I).,
```

The action of SYSACT with FUNCTION=2 after an output procedure is the same as that after an input procedure as described above; additionally the skipped character positions are filled with blank spaces. A following output procedure statement will start by handling that character denoted by character pointer position QUANTITY, which follows next to the old position of the data set.

6.3.2.3. Value of FUNCTION=3.
QUANTITY.=S.,

A SYSACT procedure statement with FUNCTION=3 is defined only if the specified data set is open and the third actual parameter QUANTITY is a variable. The procedure statement assigns the value denoted by the current position of the record pointer S to QUANTITY.

6.3.2.4. Value of FUNCTION=4.

A SYSACT procedure statement with FUNCTION=4 is defined only if the specified data set is open. In case the value of QUANTITY is less than the current position of the record pointer S, this procedure statement is defined only if QUANTITY has a value corresponding to a record pointer which has been entered and is still maintained in the internal index described in section 6.3.2.13. The action of SYSACT with FUNCTION=4 is undefined if the specified data set has been split into sections (cf. 6.1.1.4) by SYSACT with FUNCTION=8(cf.6.3.2.8) or if the data set number is 0 or 1 (cf.6.1.1.2, Footnote 5). In these cases SYSACT with FUNCTION = 14 provides similar actions.

The action of SYSACT with FUNCTION=4 depends on whether the last input/output procedure statement (not SYSACT) for the specified data set was input or output. (If input or output has not occurred since opening, i. e. since a SYSACT procedure statement with FUNCTION=12 (cf. section 6.3.2.12), input is assumed.)

6.3.2.4.1. Action After Input

S.=QUANTITY., R.=1.,

The record pointer S is set to the value of QUANTITY (1≤QUANTITY) and the character pointer R is set to 1. The data set itself is not changed. A following input or output procedure statement will start by handling the character at the first position within the record denoted by the value of QUANTITY.

Note: If the value of QUANTITY is greater than the number of records existing within the data set, the action is undefined.

6.3.2.4.2. Action After Output

```
'FOR'I.=R'STEP'1'UNTIL'P'DO'
        OUTSYMBOL (DATASETNUMBER,'('b')',1).,
'IF'QUANTITY'GREATER'S
  'THEN''FOR'J.=S+1'STEP'1'UNTIL'QUANTITY-1'DO'
        'FOR'I.=1'STEP'1'UNTIL'P'DO'
            OUTSYMBOL (DATASETNUMBER,'('b')',1)
  'ELSE'S.=QUANTITY., R.=1.,
```

The part of the current record following the item written by the last output statement is filled with blank spaces. The record pointer S is set to the value of QUANTITY (1 ≤ QUANTITY) and the character pointer R is set to 1. In case of forward skipping, i.e., if QUANTITY is greater than the current position of the record pointer S, the skipped records are filled with blank spaces. A following input or output procedure statement will start by handling the character at the first position within the record denoted by the value of QUANTITY.

Notes: After executing a SYSACT with FUNCTION=4 the programmer has to consider section 6.2.1.3.

If QUANTITY = S, i. e. if QUANTITY denotes the current record, the following occurs:

After output the free positions within the current record beyond the item written by the last ouput are filled with blank spaces. In any case the character pointer is set to the first character within the current record.

6.3.2.5. Value of FUNCTION=5
QUANTITY.=P.,

A SYSACT procedure statement with FUNCTION=5 is defined only if the specified data set is open and the third actual parameter QUANTITY is a variable. The procedure statement assigns the record length P for the specified data set to QUANTITY.

6.3.2.6. Value of FUNCTION=6
P.=QUANTITY.,

A SYSACT procedure statement with FUNCTION=6 is defined only if (1) the specified data set is closed and (2) there is not yet any data contained in it, i.e., the data set is to be created by later output procedure statements. The procedure statement specifies the record length P for that data set to be the value of QUANTITY (cf. 6.1.1.3., especially Footnote 6).

6.3.2.7. Value of FUNCTION=7
QUANTITY.=Q.,

A SYSACT procedure statement with FUNCTION=7 is defined only if the third actual parameter QUANTITY is a variable. This statement determines whether the specified data set is split into sections and sets QUANTITY to the value of Q (number of records per section) if the data set is split and to the value 0 otherwise.

6.3.2.8. Value of FUNCTION=8
Q.=QUANTITY.,

A SYSACT procedure statement with FUNCTION=8 is defined only if (1) the specified data set is closed, (2) there is not yet any data contained in it, i.e., the data set is to be created by later output procedure statements, and (3) QUANTITY has a positive 'INTEGER' type value. The procedure statement specifies that the data set to be created is to be split into sections of QUANTITY records each (cf. 6.1.1.4.).

6.3.2.9. Value of FUNCTION=9
QUANTITY.=K.,

A SYSACT procedure statement with FUNCTION=9 is defined only if the third actual parameter QUANTITY is a variable. This statement assigns the parameter K (number of blank spaces serving as a delimiter for data for the specified data set, cf. 6.2.1.4.) to QUANTITY.

6.3.2.10. Value of FUNCTION=10
K.=QUANTITY.,

A SYSACT procedure statement with FUNCTION=10 specifies that the parameter K (number of blank spaces serving as a delimiter for data, cf. 6.2.1.4.) will be the value of QUANTITY for future input or output procedure statements on the specified data set.

Note: This specification is valid only during the current execution of the ALGOL program.

6.3.2.11. Value of FUNCTION=11
QUANTITY.=C.,

A SYSACT procedure statement with FUNCTION=11 is defined only if the third actual parameter QUANTITY is a variable. This procedure statement sets QUANTITY to 1 if the specified data set is open, to 0 if it is closed and to -1 if it is exhausted (cf.6.1.1.5.), thereby determining the state of the data set.

6.3.2.12. Value of FUNCTION=12
C.=QUANTITY.,

A SYSACT procedure statement with FUNCTION=12 is defined only if the value of the third actual parameter QUANTITY is 1 or 0. This procedure statement opens the specified data set if QUANTITY=1 and the data set is close, and closes it if QUANTITY=0 and if it is not close. Otherwise the statement has no effect.

6.3.2.13. Value of FUNCTION=13

A SYSACT procedure statement with FUNCTION=13 is defined only if the specified data set is open, not split into sections (cf. 6.1.1.4.) and the third actual parameter QUANTITY is a variable. This procedure statement has two actions, an external one and an internal one.

The external action is the same as that of SYSACT with FUNCTION=3, i.e., the value denoted by the current position of the record pointer S is assigned to QUANTITY.

The internal action is to put S into an internal index which can be used by a later SYSACT procedure statement with FUNCTION=4 (cf. 6.3.2.4.) to the current record. The pointer S of any record to be retrieved through SYSACT with FUNCTION=4 by skipping backward must have been entered into this index by SYSACT with FUNCTION=13. Although it is not necessary, a record to be retrieved by skipping forward will be found more easily if it has been entered into the index.

The information entered into this index is maintained as long as the data set is open and the current record is valid, i.e., closing the data set by SYSACT with FUNCTION=12, QUANTITY=0, or a later output to any record with a record pointer position less than that of the current record destroys the entry in the index (cf. 6.2.1.3.).

6.3.2.14. Value of FUNCTION=14

A SYSACT procedure statement with FUNCTION=14 is defined only if QUANTITY>0. If the specified data set is not split into sections, the procedure statement:

```
SYSACT (DATASETNUMBER, 14, QUANTITY).,
```

is equivalent to the following sequence of statements:

```
'BEGIN' 'INTEGER'V.,
SYSACT (DATASETNUMBER, 3, V).,
SYSACT (DATASETNUMBER, 4, V+QUANTITY)'END'.,
```

The record pointer is increased by the value of QUANTITY. Therefore the remaining characters within the current record and the following QUANTITY-1 records are skipped or filled with blank spaces, depending on whether the last input/output procedure for that data set was input or output. A following transfer of data starts at the first character location of the record following those QUANTITY-1 records.

If the specified data set is split into sections of Q records each and S+QUANTITY $\leq$ Q, where S is the current value of the record pointer, the action of SYSACT with FUNCTION=14 is the same as above, i.e., if there are QUANTITY-1 records left within the current section, these records are filled with blank spaces. Otherwise, if S+QUANTITY > Q, the action is equivalent to that of:

```
SYSACT (DATASETNUMBER, 15, 1).,
```

a skip is made to the first character of the next section of the data set (cf. 6.3.2.15).

6.3.2..5. Value of FUNCTION=15

If the specified data set is split into sections of Q records each, the action of a SYSACT procedure statement with FUNCTION=15 is the following: Skip to the next section of the data set, set the record pointer the value of QUANTITY, set the character pointer to 1, and fill all character positions so skipped with blank spaces.

If the data set is not split into sections, the following two statements are equivalent:

```
SYSACT (DATASETNUMBER, 15, QUANTITY).,
SYSACT (DATASETNUMBER, 14, QUANTITY).,
```

Note: For a data set split into sections the following statement means "skip to next section (page on printer)":

```
SYSACT (DATASETNUMBER, 15, 1).,
```

6.4. Intermediate Data Storage

The pair of procedures PUT and GET permits data both to be stored temporarily on an external medium without any conversion into external formats and to be retrieved during the current execution of the ALGOL program. The data is given by "list procedures".

6.4.1. List Procedures

A procedure declaration used as list procedure must have one formal parameter specified to be a procedure which itself has one parameter; data items to be transmitted appear in the body of the list procedure as the parameter of the procedure given as the parameter of the list procedure. For example assume LIST is a list procedure, with TRANSMIT as the formal parameter. In the body of LIST, each item to be transmitted occurs as the parameter of TRANSMIT. When LIST is called by PUT or GET, internal transmission procedures will be actually substituted for TRANSMIT. The data items will be called by name and their values will be transmitted. The sequence of statements in the body of LIST determines the sequence of items in the list to be transmitted.

A simple form of a list procedure might be written as follows:

```
'PROCEDURE' LIST (TRANSMIT).,
    'PROCEDURE' TRANSMIT.,
'BEGIN' TRANSMIT (A)., TRANSMIT (B)., TRANSMIT (C)
    'END'
```

which says that the values of A, B and C are to be transmitted. A more typical list procedure might be:

```
'PROCEDURE' PAIRS (ELT)., 'PROCEDURE' ELT.,
'FOR' I.=1 'STEP'1'UNTIL'N'DO''BEGIN'
    ELT (A(/I/))., ELT(B( /I/))'END'
```

This procedure says that the values of the list of items A(/1/), B(/1/), A(/2/), ..., B(/N/) are to be transmitted, in that order. Note that if $N \leq 0$ no items are transmitted at all.

The parameter of the "transmit" procedure (i.e. the parameter of TRANSMIT or ELT in the above examples) may be an arithmetic identifier (for PUT an arithmetic expression) or a Boolean identifier (for PUT a Boolean expression), but it must be of the same type for all calls within one list procedure since identical kind and type of parameters are required. A list procedure may not execute, directly or indirectly, PUT or GET; i.e. the procedures PUT and GET may not be activated recursively. Besides, any of the features of ALGOL may be used in a list procedure.

6.4.2. Description of the Procedures PUT and GET

6.4.2.1. Assumed Procedure Declarations

```
'PROCEDURE' PUT (N, LIST).,
        'VALUE'N.,
        'INTEGER' N.,
        'PROCEDURE' LIST.,
        <procedure body>.,

'PROCEDURE' GET (N, LIST).,
        'VALUE' N.,
        'INTEGER' N.,
        'PROCEDURE' LIST.,
        <procedure body>.,
```

6.4.2.2. Semantics

The procedure PUT stores the list of values specified by the list procedure appearing as second actual parameter to an external medium and supplies the value of the first actual parameter as an identification number. Anything else previously stored with the same identification number is destroyed.

The procedure GET retrieves the list of values stored by a previous PUT, using the value of the first actual parameter as identification number. The values are assigned to the variables specified by the list procedure that appears as the second parameter. They are retrieved in the same order as they were stored and must agree in type with the variables specified by the list procedure. If fewer variables are specified than values associated with the identification number, only the first values are retrieved; if too many variables are specified, the situation is undefined.

Note: The PUT and GET must occur within the same execution of the ALGOL program.

APPENDIX 1. RELATION BETWEEN OS/360 ALGOL and ALGOL 60

In the following the restrictions of the System/360 Operating System ALGOL language with regard to ALGOL 60 as defined in the Revised Report on ALGOL 60 [1] are listed. The definitions are in terms of the ALGOL 60 report [1]

1. The own concept is not implemented:

   2.3. Delete from definition of <declarator>: "own".
   5. Delete first two sentences of fourth paragraph.
   5.1.1. Delete definition of <local or own type>:
   replace definition of <type declaration> by:
   "<type declaration> ::= <type> <type list>".
   5.1.3. Delete last sentence.
   5.2.1. Replace in definition of <array declaration> "<local or own type>"by"<type>".
   5.2.2. Delete second example.
   5.2.5. Delete this paragraph.

2. Integer labels are not allowed

   3.5.1. Delete from definition of <label> :"<unsigned integer>".
   3.5.5. Delete this paragraph.

3. Complete specification parts and equivalence of types between corresponding formal and actual parameters called by name are required.

   5.4.5. Replace third sentence by:
   "Specifications of all formal parameters if any must be supplied".
   4.7.5.5. Replace by:
   "Kind and type of actual parameters must be the same as those of the corresponding formal parameters, if called by name".

4. Only one case of letters is provided for.

   2.1. Delete from definition of <letter>:
   "a ! ... !z ! ";
   Delete"restricted, or " in the first sentence.

5. Identifiers will be differentiated only by up to six leading characters.

   2.4.3. Replace "They may be chosen freely" by:
   "Identifiers may be chosen freely, but there is no effective distinction between two different identifiers the first six basic symbols of which are common".

6. The type of an arithmetic expression will be determined in dubious cases to be real.

3.3.4. Replace the words "the following rules" of the last sentence by
"a set of rules. However, if the type of an arithmetic expression according to the rules cannot be determined without evaluating an expression or ascertaining the type or value of an actual parameter, it is real. These rules are :".

7. A goto-statement involving an undefined switch designator need not have the effect of a dummy statement.

4.3.5. Replace "equivalent to a dummy statement" by "undefined".

APPENDIX 2. REPRESENTATION OF ALGOL 60 SYMBOLS IN THE 48 AND 59 CHARACTER SETS.

ALGOL symbol (as defined in [1])	48 character set	59 character set (additional representations)
a...z	A...Z	
0...9	0...9	
+	+	
-	-	
× (multiplication sign)	*	
/	/	
÷	'/'	
↑	'POWER' or **	
<	'LESS'	<
≦	'NOTGREATER'	< =
=	'EQUAL' or =	
≧	'NOTLESS'	> =
>	'GREATER'	>
≠	'NOT EQUAL'	¬ =
≡	'EQUIV'	
⊃	'IMPL'	
∨	'OR'	!
∧	'AND'	&
¬	'NOT'	¬
, (comma)	,	
. (decimal point)	.	
10 (scale factor)	' (apostrophe)	
:	..	:
;	.,	;
:=	.= or ..=	: =
(	(	
)	)	
⌈	(/	
⌋	/)	
‘	'('	
’	')'	
␣	blank	

All other basic symbols which are represented in the ALGOL report by underlining or by boldface type as a word are punched for the System/360 Operating System ALGOL compiler as the word enclosed in apostrophes, e.g. 'TRUE', 'FALSE', 'GOTO', 'IF' and so on.

APPENDIX 3. EXAMPLES

The first two examples are complex procedure declarations. They might be separately translated and used within different ALGOL programs in the form of code procedures (cf. 5.4.6.). The third example demonstrates a program using the input/output procedures described in section 6. The further examples are possible declarations of the input/output procedures OUTREAL, ININTEGER, OUTINTEGER, INBOOLEAN, OUTBOOLEAN, OUTSTRING, reducing the definitions of these procedures to those of the procedures INSYMBOL, OUTSYMBOL, INREAL, SYSACT (cf., however, 6.1.2.).

Example 1.

```
'PROCEDURE'EULER(FCT, SUM, EPS, TIM)., 'VALUE'EPS, TIM.,
'INTEGER' TIM., 'REAL' 'PROCEDURE' FCT ., 'REAL' SUM, EPS.,
'COMMENT' EULER COMPUTES THE SUM OF FCT (I) FOR I
 FROM ZERO UP TO INFINITY BY MEANS OF A SUITABLY
 REFINED EULER TRANSFORMATION. THE SUMMATION IS
 STOPPED AS SOON AS TIM TIMES IN SUCCESSION THE ABSOLUTE
 VALUE OF THE TERMS OF THE TRANSFORMED SERIES IS
 FOUND TO BE LESS THAN EPS. HENCE ONE SHOULD PROVIDE
 A FUNCTION FCT WITH ONE INTEGER ARGUMENT, AN UPPER
 BOUND EPS, AND AN INTEGER TIM. THE OUTPUT IS THE SUM SUM.
 EULER IS PARTICULARLY EFFICIENT IN THE CASE OF A SLOWLY
 CONVERGENT OR DIVERGENT ALTERNATING SERIES.,
 'BEGIN''INTEGER' I, K, N, T., 'ARRAY' M(/0..15/).,
 'REAL' MN, MP, DS.,
  I.=N.=T.=0., M(/0/).=FCT(0)., SUM.=M(/0/)/2.,
  NEXTTERM..I. I+1., MN.=FCT(I).,
       'FOR' K.=0'STEP'1'UNTIL'N'DO'
            'BEGIN' MP.=(MN+M(/K/))/2., M(/K/).=MN.,
                MN.=MP'END'MEANS.,

      'IF' (ABS(MN)'LESS' ABS (M(/N/)))'AND'(N'LESS'15)'THEN'
            'BEGIN'DS.=MN/2., N.=N+1.,
               M(/N/).= MN'END' ACCEPT
           'ELSE' DS.=MN.,
            SUM.=SUM+DS.,
           'IF' ABS(DS)'LESS'EPS'THEN'T.=T+1'ELSE'T.=0.,
           'IF'T'LESS'TIM'THEN''GOTO'NEXTTERM
           'END'EULER
```

Example 2.

```
'PROCEDURE'RK(X, Y, N, FKT, EPS, ETA, XE, YE, FI).,
'VALUE'X, Y., 'INTEGER'N., 'BOOLEAN'FI.,
'REAL'X, EPS, ETA, XE., 'ARRAY'Y, YE., 'PROCEDURE'FKT.,
'COMMENT'.. RKINTEGRATES THE SYSTEM Y' (/K/)=
F(/K/)(X, Y1, Y2, , YN) (K=1, 2,, , N) OF DIFFERENTIAL
EQUATIONS WITH THE METHOD OF RUNGE-KUTTA WITH
AUTOMATIC SEARCH FOR APPROPRIATE LENGTH OF INTEGRATION
STEP. PARAMETERS ARE.. THE INITIAL VALUES X AND
Y(/K/) FOR X AND THE UNKNOWN FUNCTIONS Y(/K/) (X).
THE ORDER N OF THE SYSTEM. THE PROCEDURE FKT (X,
Y, N, Z) WHICH REPRESENTS THE SYSTEM TO BE INTEGRATED.
```

```
I. E. THE SET OF FUNCTIONS F (/K/). THE TOLERANCE
VALUES EPS AND ETA WHICH GOVERN THE ACCURACY
OF THE NUMERICAL INTEGRATION. THE END OF THE
INTEGRATION INTERVAL XE. THE OUTPUT PARAMETER
YE WHICH REPRESENTS THE SOLUTION AT X=XE. THE
BOOLEAN VARIABLE FI, WHICH MUST ALWAYS BE GIVEN
THE VALUE 'TRUE' FOR AN ISOLATED OR FIRST ENTRY
INTO RK. IF HOWEVER THE FUNCTIONS Y MUST BE
AVAILABLE AT SEVERAL MESH POINTS X(/0/), X(/1/),
,,,,X(/N/), THEN THE PROCEDURE MUST BE CALLED
REPEATEDLY (WITH X=X(/K/), XE=X(/K+1/), FOR
K=0,1,,,N-1) AND THEN THE LATER CALLS MAY
OCCUR WITH FI='FALSE' WHICH SAVES COMPUTING TIME.
THE INPUT PARAMETERS OF FKT MUST BE X, Y, N.
THE OUTPUT PARAMETER Z REPRESENTS THE SET OF
DERIVATIVES Z(/K/)=F(/K/) (X, Y(/1/), Y(/2/),,,
Y(/N/)) FOR X AND THE ACTUAL Y'S. A PROCEDURE
COMP ENTERS AS A NONLOCAL IDENTIFIER. THE NON-
LOCAL IDENTIFIERS S AND HS ARE ASSUMED TO BE
DECLARED 'REAL' IN A BLOCK EMBRACING ALL CALLS
OF RK.,

'BEGIN'
'ARRAY'Z, Y1, Y2, Y3 (/1..N/)., 'REAL'X1, X2, X3, H.,
'BOOLEAN' OUT., 'INTEGER' K, J.,
'PROCEDURE'RK1ST (X, Y, H, XE, YE)., 'REAL' X, H, XE.,
'ARRAY'Y , YE.,
'COMMENT'..RK1ST INTEGRATES ONE SINGLE RUNGE-
KUTTA WITH INITIAL VALUES X, Y (/K/) WHICH YIELDS
THE OUTPUT PARAMETERS XE=X+H AND YE (/K/), THE
LATTER BEING THE SOLUTION AT XE. IMPORTANT ..
THE PARAMETERS N, FKT, Z ENTER RK1ST AS NONLOCAL
ENTITIES.,

'BEGIN'
'ARRAY' W (/1..N/), A(/1..5/)., 'INTEGER' K, J.,
A(/1/).=A(/2/).=A(/5/).=H/2., A(/3/).=A(/4/).=H.,
XE.=X.,
'FOR'K.=1'STEP'1'UNTIL'N'DO'YE(/K/).=W(/K/).=Y(/K/).,
'FOR'J.=1'STEP'1'UNTIL'4'DO'
    'BEGIN'
  FKT(XE, W, N, Z).,
  XE.=X+A(/J/).,
  'FOR'K.=1'STEP'1'UNTIL'N'DO'
```

```
        'BEGIN'
        W(/K/).=Y(/K/)+A(/J/)*Z(/K/
        YE(/K/).=YE(/K/)+A(/J+1/)*(K/K/)/3
        'END'K
    'END'J
'END'RKIST.,
BEGIN OF PROGRAM..
    'IF'FI'THEN''BEGIN'H.=XE-X., S.=0'END'
    'ELSE'H  =HS.,
    OUT.='FALSE'.,
AA..'IF'(X+2.01*H-XE'GREATER'0)'EQUIV'(H'GREATER'0)'THEN'
    'BEGIN'HS.=H., OUT.='TRUE'., H.=(XE-X)/2
    'END'IF.,
    RKIST (X, Y, 2 *H, X1, Y1).,
BB..RKIST (X, Y, H, X2, Y2)., RKIST (X2, Y2, H, X3, Y3).,
    'FOR'K.=1'STEP'1'UNTIL'N'DO'
        'IF'COMP(Y1(/K/), Y3(/K/), ETA)'GREATER'EPS'THEN'
        'GOTO'CC.,
    'COMMENT'..COMP(A, B, C) IS A FUNCTION
    DESIGNATOR, THE VALUE OF WHICH IS THE
    ABSOLUTE VALUE OF THE DIFFERENCE OF THE
    MANTISSAE OF A AND B, AFTER THE EXPONENTS
    OF THESE QUANTITIES HAVE BEEN MADE EQUAL TO
    THE LARGEST OF THE EXPONENTS OF THE
    ORIGINALLY GIVEN PARAMETERS A, B, C.,
    X.=X3., 'IF'OUT'THEN''GOTO'DD.,
    'FOR'K.=1'STEP'1'UNTIL'N'DO'Y(/K/).=Y3(/K/).,
    'IF'S'EQUAL'5'THEN''BEGIN'S.=0., H.=2*H'END'IF.,
    S.=S+1., 'GOTO'AA.,
CC..H.=0.5*H., OUT.='FALSE'., X1.=X2.,
    'FOR'K.=1'STEP'1'UNTIL'N'DO'Y1(/K/).=Y2(/K/).,
    'GOTO'BB.,
DD..'FOR'K.=1'STEP'1'UNTIL'N'DO'YE(/K/).=Y3(/K/)
    'END'RK
```

Example 3.

```
'BEGIN'

'COMMENT' THIS PROGRAM GENERATES THE FIRSTTWENTY LINES OF
PASCALS TRIANGLE AND WRITES THEM TO A DATASET DENOTED BY THE
DATASETNUMBER 1.,

    'INTEGER' L, K, N, I, M, POWERTEN.,
    'INTEGER' 'ARRAY' A(/0..19/).,
    'BOOLEAN' C.,
    SYSACT (1, 6, 120)., SYSACT(1, 12, 1).,
    SYSACT (1, 4, 2)., SYSACT (1, 2, 40).,
    OUTSTRING (1, '('PASCALSbTRIANGLE')').,
    'FOR'L.=0'STEP'1'UNTIL'19'DO'
    'BEGIN'
        SYSACT(1, 4, 2*L+5)., SYSACT (1, 2, 58-3*L).,
        A(/L/).=1.,
        'FOR'K.=L-1'STEP'-1'UNTIL'1'DO'
        A(/K/).=A(/K-1/)+A(/K/).,
        'FOR'K.=0'STEP'1'UNTIL'L'DO'
        'BEGIN'
            C.='TRUE'.,
            M.=A(/K/).,
            'FOR'I.=5''3TEP'-1'UNTIL'0'DO'
            'BEGIN'
                POWERTEN.=10'POWER'I.,
                N.=M'/'POWERTEN.,
                M.=M-N*POWERTEN.,
                'IF'N'EQUAL'0'THEN'
                'BEGIN'
                   'IF'C'THEN' OUTSYMBOL (1, '('b')', 1).,
                   'ELSE' OUTSYMBOL (1, '('0')', 1).,
                'END'
                'ELSE'
                'BEGIN'
                   C.='FALSE'.,
                   OUTSYMBOL(1, '('123456789')', N).,
                'END'
            'END'
        'END'
    'END'
'END'
```

Example 4.

```
'PROCEDURE' OUTREAL (DSN, SOURCE).,
'VALUE' DSN, SOURCE.,
'INTEGER' DSN., 'REAL' SOURCE.,
'BEGIN''INTEGER' I, N, R, P, K, EXP.,
    SYSACT (DSN, 12, 1)., SYSACT (DSN, 1, R).,
    SYSACT (DSN, 5, P)., SYSACT (DSN, 9, K).,
    'IF'P-R+1'LESS'22'THEN''BEGIN'SYSACT (DSN, 14, 1)., R.=1'END'.,
    'IF'SOURCE'EQUAL'0'THEN''BEGIN''FOR'I.=1'STEP'1'UNTIL'22'DO'
                        OUTSYMBOL(DSN, '('b0bbbbbbbbbbbbbbbbbbbb')', 1).,
                        'GOTO' END
                           'END'.,
     OUTSYMBOL(DSN, '('+-')', 'IF' SOURCE'GREATER'0'THEN'1'ELSE'2).,
     SOURCE.=ABS(SOURCE).,
     EXP.=0.,
     'FOR'I.=1'WHILE'SOURCE'NOT LESS'10'DO'
       'BEGIN'SOURCE.=SOURCE/10., EXP.=EXP+1 'END'.,
     'FOR' I.=1'WHILE' SOURCE 'LESS'1'DO'
       'BEGIN' SOURCE.=SOURCE*10., EXP.=EXP-1 'END'.,
     N.=ENTIER (SOURCE).,
     SOURCE.=10*(SOURCE-N).,
     OUTSYMBOL(DSN, '('123456789')', N).,
     OUTSYMBOL (DSN, '('.')', 1).,
     'FOR'I.=14'STEP'-1'UNTIL'0'DO'
          'BEGIN'N.=ENTIER(SOURCE).,
              SOURCE.=10*(SOURCE-N).,
              OUTSYMBOL(DSN, '('0123456789')', N+1)
          'END'.,
     OUTSYMBOL (DSN, '(''')', 1).,
     OUTSYMBOL(DSN, '('+-')', 'IF'EXP'NOT LESS'0'THEN'1'ELSE'2).,
     EXP.=ABS(EXP).,
     N.=EXP'/'10.,
     EXP.=EXP-N*10.,
     OUTSYMBOL(DSN, '('0123456789')', N+1).,
     OUTSYMBOL(DSN, ' ('0123456789')', EXP+1).,
END.'FOR'R.=R+22'STEP'1'UNTIL'R+21+K'DO'
     'IF'R'NOT GREATER'P'THEN'OUTSYMBOL(DSN, '('b')', 1)
'END'OUTREAL.,
```

Example 5.

```
'PROCEDURE'ININTEGER(DSN, DESTINATION).,
      'VALUE'DSN.,
      'INTEGER'DSN, DESTINATION.,
      'BEGIN''REAL'R.,
           INREAL(DSN, R).,
           DESTINATION.=R
      'END'ININTEGER.,
```

Example 6.

```
'PROCEDURE'OUTINTEGER(DSN, SOURCE).,
'VALUE'DSN, SOURCE.,
'INTEGER' DSN, SOURCE, POWERTEN.,
'BEGIN''INTEGER' I, N, K, P, R., 'BOOLEAN'SIGN, STARTED.,
      SYSACT (DSN, 12, 1).,
      SYSACT(DSN, 5,P)., SYSACT(DSN, 1, R)., SYSACT(DSN, 9, K).,
      'IF'P-R+1'LESS'11'THEN''BEGIN'SYSACT(DSN, 14, 1)., R.=1'END'.,
      'IF'SOURCE'EQUAL'0'THEN'
           'FOR'.I.=1'STEP'1'UNTIL'11'DO'
           'BEGIN'OUTSYMBOL(DSN, '('bbbbbbbbbb0')', 1)., 'GOTO' END'END'.,
      SIGN.=SOURCE'GREATER'0.,
      SOURCE.=ABS(SOURCE).,
      STARTED.='FALSE'.,
      'FOR'I.=9'STEP'-1'UNTIL'0'DO'
      'BEGIN'POWERTEN.=10'POWER'I.,
           N.=SOURCE'/'POWERTEN.,
           SOURCE.=SOURCE-N*POWERTEN.,
           'IF'N'EQUAL'0
           'THEN'OUTSYMBOL(DSN,'('0b')', 'IF'STARTED'THEN'1'ELSE'2)

           'ELSE''BEGIN''IF''NOT'STARTED'THEN''BEGIN'STARTED.='TRUE'
                                .,OUTSYMBOL(DSN, '('+-')', 'IF'SIGN''THEN'1
                               'ELSE'2)'END'.,

                                OUTSYMBOL(DSN, '('123456789')', N)
                'END'
      'END'.,
      END..'FOR'R.=R+11'STEP'1'UNTIL'R+10+K'DO'
                'IF'R'NOTGREATER'P'THEN'OUTSYMBOL(DSN, '('b')', 1)
'END' OUTINTEGER.,
```

Example 7.

```
'PROCEDURE'INBOOLEAN (DSN,DESTINATION).,
'VALUE'DSN.,
'INTEGER'DSN., 'BOOLEAN'DESTINATION.,
'BEGIN''INTEGER'I, N, M, K, P, R.,
    SYSACT(DSN,12,1).,
    SYSACT(DSN,5,P)., SYSACT(DSN,9,K).,
    L1..SYSACT(DSN,1,R).,
        M.=0.,
        'FOR'R.=R'STEP'1'UNTIL'P'DO'
            'BEGIN''INSYMBOL (DSN,'('')', N).,
                    'IF'N'EQUAL'1'THEN''GOTO'L2
            'END'.,
       'GOTO'L1.,
    L2..'IF'R'EQUAL'P'THEN''GOTO'L1.,
       R.=R+1.,
       INSYMBOL(DSN,'('TFb')', N).,
       'IF'N'EQUAL'1'THEN''BEGIN'M.=0.,'GOTO'TRUE'END'.,
       'IF'N'EQUAL'2'THEN''BEGIN'M.=0.,'GOTO'FALSE'END.,
       'IF'N'EQUAL'3'THEN''BEGIN'M.=M+1.,
                                    'IF'M'LESS'K'THEN''GOTO'L2
                            'END'.,
       'GOTO'L1.,
    TRUE..'FOR'I.=1'STEP'1'UNTIL'4'DO'
                'BEGIN''IF'R'EQUAL'P'THEN''GOTO'L1.,
                   R.=R+1.,
                   INSYMBOL(DSN,'('RUE'b')', N).,
                   'IF'N'EQUAL'5
                   'THEN''BEGIN'M.=M+1., I.=I-1.,
                                'IF'M'EQUAL'K'THEN''GOTO'L1
                          'END'
                   'ELSE''BEGIN'M.=0.,
                                'IF'N'NOTEQUAL'I'THEN'
                                    'GOTO''IF'N'EQUAL'4'THEN'L2'ELSE'L1
                          'END'
            'END'.,
         DESTINATION.='TRUE'.,
         'GOTO'END.,
    FALSE..'FOR'I.=1'STEP'1'UNTIL'5'DO'
                'BEGIN''IF'R'EQUAL'P'THEN'' GOTO'L1.,
                   R.=R+1.,
                   INSYMBOL(DSN,'(' ALSE'b')', N).,
                   'IF'N'EQUAL'6
                   'THEN''BEGIN'M.=M+1., I.=I-1.,
                                'IF'M'EQUAL'K'THEN''GOTO'L1
                          'END'
```

```
                    'ELSE''BEGIN'M.=0.,
                                'IF'N'NOTEQUAL'I'THEN'
                                  'GOTO''IF'N'EQUAL'5'THEN'L2'ELSE'L1
                            'END'
          ' END'.,
     DESTINATION.='FALSE'.,
END..N.=0.,
   'FOR'I.=R+1'WHILE'N'NOTEQUAL'1'DO'
   'BEGIN' I.=I+1.,
       INSYMBOL(DSN,'('b')',N).,
       'IF'I'EQUAL'P'THEN''GOTO'L3
  'END'.,
  'IF'I'NOTLESS'R+K'THEN'SYSACT(DSN, 2, 1).,
L3..'END'INBOOLEAN.,
```

Example 8.

```
'PROCEDURE'OUTBOOLEAN(DSN, SOURCE).,
'VALUE'DSN, SOURCE.,
'INTEGER'DSN., 'BOOLEAN'SOURCE.,
'BEGIN ''INTEGER'I, P, R, K.,
      SYSACT(DSN, 12, 1)., SYSACT (DSN, 5, P).,
      SYSACT( DSN, 1, R)., SYSACT(DSN, 9, K).,
      'IF'P-R+1'LESS'7'THEN''BEGIN'SYSACT(DSN , 14, 1)., R.=1'END'..
      'IF'SOURCE'FQUAL''TRUE'' THEN''BEGIN''FOR'I.=1'STEP'1'UNTIL'7'DO'
                                              OUTSYMBOL(DSN,
                                              '(''TRUE'b')', I)
                                    'END'
                                'ELSE''FOR'I.=1'STEP'1'UNTIL'7'DO'
                                      OUTSYMBOL(DSN,'(''FALSE'')', I).,
    'FOR'R.=R+7'STEP'1'UNTIL'R+6+K'DO'
         'IF'R'NOTGREATER'P'THEN'OUTSYMBOL(DSN,'('b')', 1)
'END'OUTBOOLEAN.,
```

Example 9.

```
'PROCEDURE'OUTSTRING(DSN, SOURCE).,
'VALUE'DSN.,
'INTEGER'DSN.,'STRING'SOURCE.,
'BEGIN''INTEGER'I.,
        'FOR'I.=1'STEP1'UNTIL'LENGTH(SOURCE)'DO'
            OUTSYMBOL(DSN, SOURCE, I)
'END'OUTSTRING.,
```

LITERATURE

[1] "Revised Report on the Algorithmic Language ALGOL 60." Communications of the Association for Computing Machinery, volume 6 (1963), page 1.

[2] "Report on Input/Output Procedures for ALGOL 60". Communications of the Association for Computing Machinery, volume 7 (1964), page 628.

[3] "ISO Draft Proposal on the Algorithmic Language ALGOL." ISO/TC 97/SC 5 (Secr-24) 102, March 1, 1965. Appendix 1, German Proposal for representation of ALGOL symbols

[4] "ECMA Subset of ALGOL 60", Communications of the Association for Computing Machinery, volume 6 (1963), page 595.

[5] "Report on SUBSET ALGOL 60 (IFIP)," Communications of the Association for Computing Machinery, volume 7 (1964), page 626.

[6] D. E. Knuth, "A list of the remaining trouble spots in ALGOL 60", ALGOL Bulletin 19 (1965), page 19.

INDEX

(The numbers denote the sections of this paper and the corresponding pages)

Printed in U.S.A. C28-6615-0

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601

READER'S COMMENTS

tle: IBM System/360 Operating System
ALGOL Language

Form: C28-6615-0

he material:	Yes	No
Easy to Read?	___	___
Well organized?	___	___
Complete?	___	___
Well illustrated?	___	___
Accurate?	___	___
Suitable for its intended audience?	___	___

low did you use this publication?
___ As an introduction to the subject ___ For additional knowledge
Other _____

fold

Please check the items that describe your position:

___ Customer personnel ___ Operator ___ Sales Representative
___ IBM personnel ___ Programmer ___ Systems Engineer
___ Manager ___ Customer Engineer ___ Trainee
___ Systems Analyst ___ Instructor Other_____

Please check specific criticism(s), give page number(s),and explain below:
___ Clarification on page(s)
___ Addition on page(s)
___ Deletion on page(s)
___ Error on page(s)

Explanation:

fold

Name _____

Address _____

FOLD ON TWO LINES,STAPLE AND MA
No Postage Necessary if Mailed .A.

fold

FIRST CLASS
PERMIT NO. 81

POUGHKEEPSIE, N.Y.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

IBM CORPORATION
P.O. BOX 390
POUGHKEEPSIE, N. Y. 12602

ATTN: PROGRAMMING SYSTEMS PUBLICATIONS
DEPT. D58

---

fold

Printed in U.S.A. C28-6615-0

IBM
International Business Machines Corporation
Processing Division
[illegible] N.Y. 10601

sta