KONINKLIJKE MILITAIRE SCHOOL ECOLE ROYALE MILITAIRE

A L G O L 68 / 19

R E F E R E N C E M A N U A L

by

P. E. GENNART
PROFESSOR

AND

G. LOUIS

1 9 7 5

F. E.

UNIVERSITE CATHOLIQUE DE LOUVAIN

CENTRE DE CALCUL

A L G O L 68/19

REFERENCE MANUAL

MARS 1976

G. LOUIS

0.1. Introduction to version 1, november 1973.

ALGOL 68 is defined by the Report on the Algorithmic Language ALGOL 68 (A. van Wijngaarden (Ed), Numerische Mathematik, 14, 79-218, 1969), which will be referred to as "the Report".

ALGOL 68/19 is a subset of ALGOL 68, formally defined by a set of rules modifying the Report. It's also called ALGOL72.

This Reference Manual is intended as a complete, but informal description of the language, together with information about its implementation on an IBM360/30 computer, operating under DOS. Most of the implementation information is given in Appendices.

Appendix 2 contains a context-free grammar for the language ; it may be very useful for the programmer as a short syntactic definition. A somewhat modified use of the formalism of Appendix 2 is made in this manual to give the so-called "general constructions" of syntactic notions.

The technical terms used in this Manual are put between "quotes" (") at their first appearance, or at some important use. They sometimes differ from the terms used in the Report. They will not always be defined.

Some outstanding characteristics of ALGOL 68/19 and of its compiler are :

- the mode constructions of ALGOL68 are included, with the exception of unions and structures ; no mode definition mechanism is provided ;

- no operator definition mechanism is provided ;

- the method of passing parameters is that of ALGOL68 ;

- all procedures are 'void', i.e. there are no 'functions' ;

- input-output is not that of ALGOL68, but there is a complete set of physical input and output routines, not using the DOS- system ;

- no formats are implemented, but a set of conversion routines is provided to and from character strings.

- runtime errors are checked with localization and 'traceback' ; in particular, bounds of multiple values are checked ;

- storage organization is dynamic (no 'heap') ;

- the possibility of separate compilation of procedures is provided.

The design and the implementation of ALGOL 68/19 are the work of Guy LOUIS and Piet RYPENS. They received a valuable programming aid from A.M. DOYEN and A. MICHAUX.

0.2. Introduction to version 2, march 1974.

The success of the first version of ALGOL 68/19 has led us to introduce some additions in order to facilitate the task of the programmer. The most important additions are the introduction in the standard prelude of mathematical functions (which are no longer void), and of formatted input-output on the standard devices, the extension of the comparison operators eq ,... to character operands, the optional suppression of the redundant repetition of the virtual parameters in a procedure declaration and the addition of a case-statement. The DOS-source statement library can be use to include and update parts of programs.

A very fast link-and-go compiler has been implemented, with the only restriction that no separately compiled procedures are permitted. An overlay feature is also available. A set of plotting routines has been included in the library prelude.

0.3. Introduction of version 3, november 1975.

ALGOL 68/19 has been implemented under VM/CMS on an IBM 370/158 computer; this was the work of Guy LOUIS.

Some modifications have been introduced to the language, such as:

- optional suppression of "apostrophes" (') surrounding "keywords", with the restriction that keywords may no longer be used as identifiers;

- modes of separately compiled procedures are now tested for compatibility with pragmats;

- addition of an original "common" feature;

- addition of a powerfull conversational "trace" at runtime;

- complete insertion in the VM/CMS virtual system;

- possibility of calling "FORTRAN" subprograms; ...

1. Examples of simple programs.

a. Card-to-tape program.

```
EX01: begin [1:80] char BUF ;
          do (TAKE(BUF,1,80,EOF);  # repetition ; card input routine #
              TAPEW(2,BUF,1,80)    # tape output #
             )                     # EOF is standard end-of-file #
      end
```

b. Card-to-tape program, with blocking, and end-of-file mark.

```
begin int BL = 10; # blocking factor #
      int DIM = 80*BL; # dimension of record #
      [1:DIM] char BUF; int N;
       # this procedure replaces the standard routine EOF: #
      proc EOF=(: begin  if N  gt 1
                         then BUF[1+(N-1)*80]:= repr 255;
                              # logical end of file = X'FF' #
                              TAPEW(2,BUF,1,DIM)
                         fi ;
                         WTM(2);STOP
                   end
                );
      do ( for I  to BL
           do (N:=I;TAKE(BUF,1+(I-1)*80,80,EOF));
           TAPEW(2,BUF,1,DIM)
         )
end
```

c. Quadratic equation solver.

```
PROG: begin real A,B,C,X1;
            GETR(A); GETR(B); GETR(C); #simplified input#
            if A  eq 0
            then PUTS("a is zero"); goto PROG
            else  begin B:=B/A;C:=C/A;
                        real D=B**2-4*C;
                        real SQT;
                        if D  lt 0
                        then PUTS("no real roots"); goto PROG
                        else
                              SQT:=SQRT(D);
                              if B  gt 0
                              then X1:=(-B-SQT)/2
                              else X1:=(-B+SQT)/2
                              fi ;
                              PUTR(X1); PUTR(C/X1)
                               # simplified output #
                        fi
                  end
            fi ; goto PROG
     end
```

d. Recursive calculation of a greatest common divisor.

```
PGCD: begin  proc PGCD =
                  (( int A,B): if B  eq 0  then PUTI(A)
                                           else PGCD(B,A mod B)
                                            # PGCD calls itself #
                               fi
                  );
            PGCD(12,8)
      end
```

e. Updating records on CMS-files.

```
begin
    [] char FILE = "TEST DATA A1"; #CMS identification of the FILE#
    int LENGTH = 80; #length of records in the FILE#
    int N; #number of the records to be updated#
    [1:LENGTH] char BUFFER;
    FORMAT(1,"*(I5,L,S80)");
        # a format is available on the virtual reader #
    proc EOF1 = (: begin CLOSE(FILE); STOP end );
        # end of file procedure for the reader #
    ON(77,EOF1); # recuperate end of file on reader #
    proc EOF2 = (: begin PUTS("end of file on disk"); EOF1 end );
        # end of file procedure on the FILE #

    do (
         GET(N); # read number of the record to be updated #
         DISKR(FILE,N,BUFFER,1,LENGTH,EOF2);
                 # read the N-th record on the FILE #
         DISPLY(BUFFER,1,LENGTH); # displays on terminal #
         GET(BUFFER); # read the record to be updated #
         CLOSE(FILE); # close the input file; now,
                        it's used for output #
         DISKW(FILE,N,BUFFER,1,LENGTH);
         DISPLY(BUFFER,1,LENGTH)
       ) # continue the loop #
end
```

f. Towers of hanoi.

```
begin
    proc hanoi = (( int PEG1, PEG2, NUMBER ):
    begin int WORK = 6-PEG1-PEG2; %compute workpeg%
          if NUMBER gt 0
          then HANOI(PEG1,WORK,NUMBER-1);
               FORMAT(3,"S14,I2,S8,I2,S7,I2");
               PUT("move from peg",PEG1,"to peg",PEG2,"piece",NUMBER);
               HANOI(WORK,PEG2,NUMBER-1)
          fi
    end
    );

for I to 5 do
  ( PAGE; PUTI(I); LINE(3); HANOI(1,2,K)  )
end
```

g. Inner product.

```
INNER:
  begin
       int RES;
       proc PROD =
        (( [] real MAT1, MAT2, ref real RES ):
          if lwb MAT1 ne lwb MAT2 or upb MAT1 ne upb MAT2
          then PUTS("......"); RES:=-32768
          else RES:=0;
               for I from 1 lwb MAT1 to upb MAT1
               do ( RES:=RES+MAT1[I]*MAT2[I] )
          fi
        );

       int N, M; GETI(N); GETI(M);
       [N:M] real MAT1, MAT2;
       PROD(MAT1,MAT2,RES); PUTR(RES);
       [1:10] real MAT; FORMAT(1,"*(F10.0)"); GET(MAT);
       PROD(MAT,MAT,RES); FORMAT(3,"E14.7"); PUT(RES)
  end
```

2. Symbols.

ALGOL 68 "programs" are written with "symbols". Each symbol possesses one or more "representations", which may vary from one implementation to another. The symbols used in ALGOL 68/19 are given in appendix 8, together with their representations, to be used as input for our compiler, and some other representations used in printed texts.

Some symbols are single characters (letters, digits, special characters). Compositions of single characters may constitute one single symbol, e.g. ':=' is the "becomes-symbol". Other symbols are represented by strings of characters in capital letters, enclosed or not between apostrophes (for punching) or in underlined lower case letters, or in boldface type (for printing). An example is begin, or 'BEGIN', or BEGIN, the "begin-symbol".

Unlike ALGOL 68, ALGOL 68/19 does not provide for the definition of new symbols.

3. Meaning of a program.

The meaning of a program is explained in terms of a hypothetical computer which performs a set of "actions", the "elaboration" of the program. The computer deals with a set of "objects" between which, at any given time, certain "relationships" may 'hold".

Relationships are either "permanent", i.e. independent of the program and its elaboration, or actions may cause them to hold or to cease to hold.

Relations are between objects, which can be either "external" or "internal".

External objects appear in the program, and are strings of symbols.
Examples : ;) 14 ALPHA X + Y

Internal objects are "values".

An external object may "possess" a value (this is one of the relations which may hold between objects).

Examples : 14 possesses the suggested value,
; possesses no value.

The various kinds of values are described in the next section.

4. Values.

Some values may be considered as elementary ; they are subdivided as follows :

"Plain values" are :

"arithmetic values", i.e "integer" numbers or "real" numbers,
"truth values",
"characters".

"Formats", as values, are not implemented in ALGOL 68/19, but a formatted input-output is available (see A.10).

Other values are constructed with the aid of elementary values.

They are : "multiple values" (corresponding to arrays in other languages) ; they will be described later ;
"structured values" exist in ALGOL 68, but not in ALGOL 68/19 ;
"names" and "routines" will also be described later.

Each arithmetic value has a "length number", which characterizes the precision of its representation in storage. The number of different length numbers may vary from one computer to another. We retain only two different length numbers, corresponding to the available precisions on an IBM/360 or 370 computer. Possible relations between arithmetic values will be defined later ; they correspond to the usual mathematical relations.

Truth values are suggested by their denotations :

true and false .

Character values are defined in each implementation by a set of available characters (see A.4.).

5. Modes.

Each value is of one specific "mode". A mode may be seen as the common characteristic of a set of values. A mode can be "specified" in a program by an external object called a "declarer". In this text, we will also use terms taken from the syntax of the Report; some of these terms are given below, together with declarers which specify modes.

Value	Declarer	Mode
integer	int or long int	"integral" or "long-integral"
real	real or long real	"real" or "long-real"
truth	bool	"boolean"
character	char	"character"
routine	begins with proc	begins with "procedure"
multiple	begins with [] or [,] or ...(1)	begins with "row-of" or "row-of-row-of" or ...(2)
name	begins with ref	begins with "reference-to"

(1) a number of comma-symbols between [and]
(2) a number of times "row-of"

6. Denotations.

"Denotations" correspond to constants in other languages.

They are external objects which possess permanent values, not dependent upon the elaboration of the program.

Limitations on the values of denotations are to be found in A.4.

6.1. Integral denotations.

Examples : 0 4096 00123 long 0

Note : -1 is not a denotation, but a monadic formula (see 13).

6.2. Real denotations.

Examples : 0.000123 1.23E-4 long 3.141592635
.123 12E25

Note : 1. is not a denotation, but .1 is a real denotation.

6.3. Boolean denotations.

true and false

6.4. Character denotations.

Each character denotes itself, except the "quote-character", which is denoted by a double quote-symbol. The denotation is placed between two quote-symbols.

Ex: "A", " ", """", c, $$$$.

6.5. String denotations.

A character string is not an elementary value ; its mode is 'row-of-character' ; it has nevertheless a denotation.

Examples : "ABC"
$A=$$a$$$ (the value has five characters)
"A B + C" (the value has seven characters)
"" or $$ (empty string)

Notes : 1. A character string denotation can only denote a value with 0, or 2, or more characters. If there is only one character between the quotes, it is a character denotation.

2. The "space-character" is represented in a string by itself. At all other places, spaces may be freely used.

6.6. Routine denotations. (see 21)

7. Spaces and comments.

"Spaces" may be inserted between "symbols". They are also used as "delimiters" between "identifiers" and "keywords". They have no special meaning outside "strings". A space in a string is a "space-symbol".

"Comments" may be inserted everywhere in a program. They are enclosed between two "comment-symbols". Four representations are given for this symbol: #, ¢, pr and co. pr and co will be used when the comment is intended to give information to the compiler at the beginning of a program (see 21.4 and 21.5). # and ¢ denote the standard comment symbol.

Examples:

```
pr proc(int) EOF1 pr # EOF1 is an external procedure #
co NAME1 co ¢ NAME1 is a labeled common ¢
# this mini-program calls EOF1: #
begin EOF1(-7) end
```

8. Identifiers and identity-declarations.

"Identifiers" are external objects which may possess values.

Examples of identifiers : X
A1
MA#THU#SA#LEM# (1)

The first symbol of an identifier must be a letter, the next symbols may be letters or digits. In our implementation, the total number of symbols may not exceed 6, but see (1).

Note also that identifiers cannot be choosen to be "keywords".

Unlike denotations, which possess permanent values, mode identifiers are made to possess values by the elaboration of "identity declarations". Examples :

real A = 3.14 ;

real X ;

These two examples show the two possible syntactic forms of an identity declaration. The former contains an "equal-symbol".

In ALGOL 68/19, this is the only place where the equal-symbol has the representation '='. The second construction will be described in section 9. For the former the general construction is

<virtual declarer> <identifier> = <expression>.

(for procedures, there is a special rule ; see 22)

The word "virtual" will be explained in connection with multiple values (section 20). It has no special meaning outside that case.

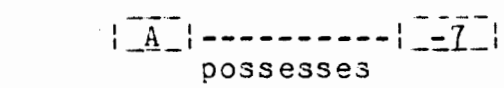
The declarer specifies some mode (see 5) ; the expression (see 13) must yield a value of the same mode. The simplest case of an expression is a denotation, as in the example above.

After elaboration of the first declaration, the identifier A possesses the value of 3.14, and that value cannot be changed. It is in fact a "constant", represented in the program (or in part of it - see 27) by an identifier. "Variables" will be described in the next section.

After reading section 17, you will understand that the position at the right of '=' is strong, which means that some automatic changes of mode are possible there.

The following sketch represents the process of the elaboration of int A = -7 :

identifier integral value



Other examples of valid declarations of the first kind are :

```
real PI = 3.141592 ; real PI2 = PI/2 ;
long int ONE = long 1 ;
proc NO#THING# = ( : skip )
```

9. Names, and associated declarations.

9.1. Names.

"Names" are values (internal objects), which may be seen as memory locations or addresses. There are of course no denotations for names, but there must be some possibility to "create" names (this involves the reservation of memory), and to associate an identifier with it. This is achieved by a declaration of the second form, described hereafter.

9.2. Creation of names.

Example : real X ;

The general construction of a declaration of this kind is :

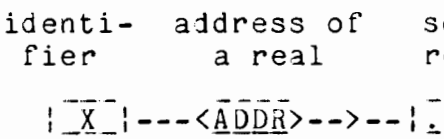
<actual declarer> <identifier> , or more generally,

<actual declarer> <identifier>{,<identifier>}* (1)

The process of the elaboration of the declaration real X may be summarized as follows :

Memory space is allocated for a real number ;
some real number is registered in it ;
the identifier X is made to possess the name
(the address) of the allocated memory location.

The sketch represents this process.



More technical terms to describe the same process are : a name of the mode 'reference-to-real' is created ; that name is made to

"refer to" some real value ; the identifier X is made to possess that name.

To make a name refer to a given value, an assignation may be used (Example : X := 3.14 ; see 12).

Notes : 1. The triplet identifier-name-value corresponds to the concept of a variable in other languages.

2. A name is always 'specialized', i.e. it refers to a value of a given mode.

3. To have the mode of a name, add 'reference to' before the mode specified by the declarer in the declaration which creates that name.

4. "to refer to" is a relation between two internal objects.

An example of construction (1) :

real X, Y, Z; is equivalent to

real X ; real Y ; real Z ;

10. Deleted.

11. The program.

11.1. General construction.

The simplest general construction of a "program" is

[<label> :] <block> ,

where a "label" is an identifier (if omitted, the label is considered to be 'MAINPG') and a "block" has the following general construction:

begin <serial clause> end

and a <serial clause> :

{ { <statement> ; }* <declaration> ; }*
[<label> :]<statement> { ;[<label>]<statement> }*

In other words, a program is an optional label, followed by a block. The block itself is enclosed between a begin-symbol and an end-symbol, and is composed of some number of declarations (see 8 and 9) (this number may be 0), possibly interspersed with unlabeled statements, and followed by at least one possibly labeled statement.

The allowed "statements" are :

a "jump" : goto <label> (see 11.4)
a "null-statement" : skip
a "block" (see above)
an "assignation" (see 12)
a "call" (see 21)
a "conditional statement" (see 18)
a "repetitive statement" (see 19)

The embedded block structure of ALGOL 68/19 follows from the fact that a block is a statement (and can be a constituent of another block).

"Common" and "pragmats" can also be used before or after a program (see 21.4. and 21.5.).

11.2 Deleted.

11.3. Elaboration of a program.

The "elaboration" of a program is the elaboration of its block, i.e. the "serial" elaboration of its declarations and statements (serial means in the order in which they are written). This serial elaboration may be altered by the elaboration of a jump (see 11.4).

11.4 Jumps.

The label of a jump (see 11.1) must be the label of some statement.

The elaboration of a jump consits only in an alteration of the serial elaboration of the statements of a block : the next statement to be elaborated will be the statement which is preceded by the same identifier as that of the jump. This statement may be outside of the block (see 27.2).

Example : L : goto L (never write this)

11.5. Skips.

A skip may sometimes be required to place a label somewhere.

```
Example:   do ( ....
                if COND then goto END1
                        else ....
                        fi;
                ....
                END1: skip
             )
```

12. Assignations.

An assignation is a statement, the elaboration of which causes a name to refer to some value.

Example : X := 1.0

In this example, the value possessed by the "source" 1.0 is assigned to the name which is the value possessed by the "destination" X.

The general construction is :

<destination> := <source>

The destination and the source are "expressions" (see 13).

Note : In Appendix 2, the destination and the source are given as 'expressions'.

The mode of the destination has to be 'reference', followed by the mode of the source, so that the name possessed by the destination can be made to refer to the value of the source. In the example above, the mode of 1.0 is 'real', so that the mode identifier X must have been declared 'reference-to-real' (see 9 and 27.2), for example by real X ;

In the example, after the elaboration of the assignation, the memory location corresponding to X contains the arithmetic value 1, in floating point internal representation, single precision.

Some automatic changes of mode are possible during the elaboration of an assignation ; they are known as "coercions" (see 17).

13. Expressions.

Expressions are external objects possessing values.

An expression can be :	Examples :
an identifier (see 8)	ALPHA
a denotation (see 6)	1.345 , true , "ab"
a "slice" (see 20.3)	X1[K]
a "formula" (see 14)	(X+1-Y)/2.5
an expression between parentheses	(A+B)
a call of a mathematical function	SIN(A+B)

14. Formulas.

"Formulas" are used to instruct the computer to perform operations in the ordinary sense.

Formulas are either "dyadic formulas" or "monadic formulas".

A dyadic formula is one "operator" between two "operands".

Example : Y-X.

A monadic formula is one operator, followed by one operand.

Example : -X.

Operands may be any expression (see 13).

Because a formula may be an operand of another formula, composite formulas are possible, like X+Y*Z, or X+Y+Z.

The elaboration of a formula consits in the elaboration of the operands, followed by performing the operation. In composite formulas like the last two examples, the point is to know the order of elaboration, i.e. which formulas are to be considered as operands for another one.

The answer is given by the "priority" rules. Each operator has one own priority. The operators available in ALGOL 68/19 are given below, together with their priorities. They are described in the following section (15). The effect of the priority rules will be described in section 16.

```
priority  1 :  none in ALGOL68/19
priority  2 :  or
priority  3 :  and
priority  4 :  eq  ne
priority  5 :  lt  le  ge  gt
priority  6 :  - + (dyadic)
priority  7 :  * /  over  mod
priority  8 :  **  upb  lwb
priority  9 :  none in ALGOL68/19
priority 10 :  (monadic operators)
               not - +  abs  repr  upb  lwb
               leng  short  odd  sign
               round  entier
```

15. Description of the operators.

"Operators" operate on "operands" with given modes, and yield results with given modes.

15.1. Boolean operators.

```
not   ("not", monadic, priority 10)
and   ("and", dyadic, priority 3)
or    ("or", dyadic, priority 2)
```

are the usual boolean operators, operating on boolean values, and giving a boolean result.

15.2. Comparison operators.

```
eq    ("equal", dyadic,priority 4)
ne    ("not equal", dyadic, priority 4)
lt    ("less than", dyadic, priority 5)
le    ("less or equal", dyadic, priority 5)
ge    ("greater or equal", dyadic, priority 5)
gt    ("greater than", dyadic, priority 5)
```

are the usual comparison operators. They operate on arithmetic or character operands, and give a boolean result.

If the operands are arithmetic, both must be of the same length, but each of them may be either integral or real.

Examples : 1) X lt 5
2) the value of 1 lt 5 is that of true
3) [1:N] char A ; ... ; if A[1] eq "Z" ...

To compare strings, use the procedure COMP (Appendix 3).

15.3. Arithmetic operators.

a) '+' and '-' ("plus" and "minus", dyadic, priority 6)
'*' and '/' ("times" and "divided by", dyadic, priority 7)

are the arithmetic operators usually available by hardware on the computers,with arithmetic operands, and arithmetic results.

Both operands must be of one same length, and the result is of that same length. Each operand may be either integral or real.
The mode of the result of '/' is always '(long-)real'.
The mode of the result of '+' , '-' or '*' is also '(long-)real', unless if both operands are of the '(long-)integral'mode, in which case the result is also of that same mode.

Examples : A+1
B*A

b) + and - ("plus" and "minus", monadic, priority 10)

are the usual monadic operators on arithmetic operands.

'+' has no effect ; '-' changes the sign

Examples: -X
-1
-(X+1)
+X

c) '**' ("up" dyadic, priority 8)

is the exponentiation operator.

Example : A**5

The second operand must be integral (not long-integral).

The result has the mode of the first operand.

The elaboration of such a formula consits in successive multiplications, followed by inversion, if the second operand is negative.

There is no operator to raise a real value to the power another real value. Standard arithmetic functions should therefore be used. (see 24).

15.4. Special operators.

a) over and mod ("over" and "modulo", dyadic, priority 7)

operate on integral operands of one same length, giving a result of the same length.

The result of A over B, is the result Q of the arithmetic division of A by B, defined by

|A| = |B|*|Q|+R, B ne 0, A*B*Q ge 0, 0 le R lt |B|

The result of A mod B is the value of RES = A-(A over B)*B, if RES is not negative; otherwise, it is RES+|B|.

Examples :

The result of			is ...
10	over	3	3
10	over	-3	-3
-10	over	3	-3
-10	over	-3	3
10	mod	3	1
10	mod	-3	1
-10	mod	3	2
-10	mod	-3	2

b) upb and lwb (monadic, priority 10
(dyadic, priority 8)

(see 20.4., multiple values)

c) leng and short ("lengthen" and "shorten", monadic, priority 10)

The operand has the mode int or real .

- leng, operating on an operand of the mode int or real , gives a result of the mode long int , resp. long real , the value of which is the equivalent value of the operand (to be "equivalent" is a relation which can hold between two values, see 3).

- short, operating on an operand of the mode long int or long real , suppresses a long from the mode of the operand. If the operand has the mode long int , the result is its equivalent short value, if any. If the operand has the mode long real , then the result is the number of the real mode whose value lies the nearest that of the operand.

Examples : leng x + long 1

leng is an operator,
long is part of the long-integral-denotation
long 1

short long 123456789876 does not exist in our implementation, because there is no short equivalent of 123456789876.

d) odd ("odd", monadic, priority 10)

The operand has the mode int or long int .

The result is of the mode bool , and has the value of true

if the operand is odd, else it has the value of false .

e) sign ("sign", monadic, priority 10)

The operand has the mode int , or long int
or real or long real

The result is of the mode int , and has the value:

-1	for a negative operand
0	for zero
+1	for a positive operand.

f) round ("round", monadic, priority 10)

The operand is real or long real .

The result is resp. int or long int .

This is the rounding operation. The value of the result must not differ from the value of the operand by more than 0.5. The value of round X is that of entier (X+0.5).

It should be noted that the conversion from real to integral is always explicit (see also the next operator, entier).

g) entier ("entier", monadic, priority 10)

The operand is real or long real

The result is resp int or long int . Its value is the greatest integer the value of which is not greater than that of the operand.

Examples: the result of entier 1.5 is 1
the result of entier -1.5 is -2
the result of entier 2.0 is 2

h) abs ("absolute value of", monadic, priority 10)

1. If the operand is a character, then the result is an integral value (mode int), the "integral equivalent" of the character (this is a machine-dependent result, see Appendix 4).

2. If the operand has an arithmetic value, then the result is an arithmetic value of the same mode, the absolute value of the operand.

i) repr ("representation of", monadic, priority 10)

is the inverse operator of abs operating on a character.

The operand A is an integral value (mode int) ; the result is the character X, if any, for which abs X = A.

In our implementation, repr gives a result which can be internally stored for every value of the operand from 0 to 255. Output of some of these characters may be impossible on some devices, without however giving rise to an error condition (see Appendix 4, giving the available printer characters).

16. The elaboration of formulas.

The effect of the priority rules in a formula may be summarized as follows :

If an operand is parenthesized, or if it is an identifier, a denotation, a subscripted identifier, or a call, then it is 'easily identified'.

If this is not the case, the priority rules are used to identify the operands. The parenthesizing rules which follow may be used for this purpose.

Step 1. Identify all the monadic operators.
An operator is monadic if it is at the left of an easily identified operand, and preceded by another operator, or if it begins a formula. When a monadic operator is found, put parentheses around it and the next easily identified operand, then take step 1 again, otherwise, take step 2.

Example :

```
-X+Y becomes (-X)+Y
X--Y becomes X-(-Y)
X--Y+-Z becomes X-(-Y)+(-Z)
```

Step 2. To identify the operands of dyadic operators consider the leftmost operator of the highest priority, if any, not contained in an easily identified operand, and put parentheses around it and its two neighbouring easily identified operands, then take step 2 again.

Example :

```
X+Y-Z          becomes (X+Y)+Z
X+Y*Z          becomes X+(Y*Z)
X+Y*Z**2       becomes X+(Y*(Z**2))
X+Y[I+J+K]*2   becomes X+(Y([I+J+K]*2)
```

At the end of the process, the expression is fully parenthesized, and all operands are easily identified. The elaboration begins with the elaboration of the operands of the outermost formula (this may involve elaboration of inner formulas, with additional parenthesizing processes if needed, as in the last example above), followed by the elaboration of the operation itself.

These rules follow from the syntax of Appendix 2, where

<Ol> stands for a formula.
<Opi> stands for an operator with priority i (i=1,2....10).

It follows from the rule

<Oi> ::= <Oi> <Opi> <Oi+1>

that a formula like X+Y-Z will be elaborated from left to right.

Note that -1**2+4 has the value 5,
whereas 4-1**2 has the value 3.

17. Coercions.

a. "Coercions" are automatic changes of mode, which are part of the elaboration of a program. These changes affect the values of expressions.

Example : In the identity declaration real X= 1, the mode of X is 'real', whereas the mode of 1 is 'integral'. Before making X to possess its value, the mode of 1 has to be changed from 'integral' to 'real'. This is a coercion, known as "widening".

Each coercion transforms the "a priori" mode of an expression into an "a posteriori" mode.

b. The only coercions available in ALGOL 68/19 are "widening" and "dereferencing".

When widening, the a priori mode is 'integral' or 'long-integral', and the a posteriori mode is 'real' or 'long-real', respectively.

The value after coercion is the real value which is "equivalent" to the integral value before coercion. Every integral value of one length must of course have an (exact) equivalent real value of the same length.

Dereferencing suppresses a 'reference-to' before the mode of the given value. Dereferencing is necessary, e.g. in the assignation

X := Y

if both X and Y are of the 'reference-to-real'mode, because it is the value to which Y refers that is needed in the source, so a 'reference-to' has to be removed from its mode : it goes from the a priori 'reference-to-real'mode to the a posteriori 'real' mode.

c. Coercions are not permitted in all "positions" of expressions. Positions are characterized by the words : "strong", "firm", or "weak".

A weak position is that of the identifier of a slice (e.g. X1 in X1[K] ; see 20.3).

A firm position is that of an operand in a formula (see 14).

Strong positions are those of :

sources in assignations (see 12),

expressions in declarations (see 8),

bounds in declarations of multiple values (see 20)

subscripts (see 20),

expressions in repetitive statements (see 19),

the integral expression in a case statement (see 18),

the boolean expression in a conditional statement (see 18),

expressions used in calls (see 21.c),

identifiers of calls (see 21).

Note. For some of the positions given as strong, a 'firm' would have been sufficient. They are however given as strong to avoid departing from the Report.

d. The allowed coercions are :

- in a weak position : a number of times dereferencing, leaving at least one ref ;
- in a firm position : a number of times dereferencing ;
- in a strong position : a number of times dereferencing, possibly followed by a widening.

Examples :

```
real X ;
ref ref ref real XXX ;
ref ref int III ;
```

in X := 1 , 1 is widened to real in its strong position ;

in X := III , III is dereferenced three times, then it is widened ;

in XXX + 1 , XXX is dereferenced four times, to yield a real value ; the operand 1 is not widened (this is not permitted in the firm position of an operand), but the operation + has been defined also when one operand is real and the other one integral.

16. Conditional statements.

16.1. If-statement.

Examples: 1.

```
if D lt 0 then D:=0 fi ;
```

2.

```
if D lt 0
then PUTS ("NO ROOTS") ;
     STOP
fi
```

3.

```
if D lt 0
then D:=0; PUTS("no roots")
else D := SQRT(D) ;
     PUTR (-B+D)/(2*A) ;
     PUTR (-B-D)/(2*A)
fi
```

The general construction of an "if-statement" is :

```
if < expression> then <statement>{;<statement>}*
                { else <statement>{;<statement>}*}0
fi
```

The entire statement is enclosed between the two special brackets if and fi .

The expression must yield a boolean result ; its position is strong (see 17).

The then -part is mandatory ; the else -part in optional.

The elaboration of a conditional statement consits of the following steps :

- elaborate the expression ;
- if its value is true , elaborate the statements of the then -part and skip the statements of the else -part, if any.
- if its value is false , skip the statements of the then -part and elaborate the statements of the else -part, if any.

Note that only statements may be conditional, whereas in ALGOL 68 expressions also may be conditional.

16.2. Case-statement.

The "case-statement" provides an alternative for some compound if-statements which could otherwise look like:

```
if c1 then s1
        else if c2 then s2
                   else if ...
```

The general construction is

```
case <expression>
in <statement1>,
   <statement2>,    ...
   <statement N>
{ out<statement N+1> }0
esac
```

The expression must yield an integral result. Its position is strong.

The in -part must contain at least one statement.

The elaboration of the case-statement begins with the elaboration of the expression. If its value is I, with 1 le I le N, then statement I is elaborated; otherwise, statement N+1 is elaborated.

Note that out skip esac is equivalent to esac .

Example:

```
proc SWITCH = (( int N):
      case N in goto L1 , goto L2 , goto L3
      out STOP
      esac );
int I ; ... SWITCH(I) ;
```

19. Repetitive statement.

Examples. 1.

```
for I from 10 by -2 to -5 do
      ( A[I]:= A[I]+ 1 ;
        B[I]:= B[I]+ 1
      )
```

2.

```
for I while A+I lt 10 do (PUTI(I))
```

The general construction of the "repetitive statement" is :

{ for <identifier> }0 { from <expression> }0
{ by <expression> }0 { to <expression> }0 { while <expression> }0
do (<serial clause>) (see 11.1.)

The identifier following for must be considered as being declared there with the mode 'integral' ; it follows from this that the programmer cannot change its value.

The first three expressions must yield integral values, possibly after some coercions, because the position is strong. The expression after while must yield a boolean value.

Semantics.

If all parts of the statement are present, the semantics are described by the following equivalence :

```
for I  from <E1>  by <E2>
       to <E3>  while <E4>  do
       ( <CHAIN> )
```

is equivalent to

```
begin   int J;
        J := <E1> ;
        int K = <E2> ;
        int L = <E3> ;
        bool B ;
M :  if K  gt 0
     then  B := J  le L
     else  if K  lt 0
           then B := J  ge L
           else B :=  true
           fi
     fi ;
     if B
     then   begin     int I = J ;
                      if <E4>
                      then  <CHAIN> ;
                            J := J + K ;
                            goto M
                      fi
            end
     fi
 end
```

The for I -part is required only when the value of I is used in the chain or in the expression after while .

If the from -part is omitted, the effect is the same as with from 1.

If the by -part is omitted, the effect is the same as with by 1.

If the to - part is omitted, no test will be provided by that part on the value of I.

The while - part may be used to introduce some other test to end the loop ; it may use the identifier of the for -part.

The effect of a for - from - by - to - part may be summarized as follows :

If the identifier has a value which does not exceed (see below) the value of the third expression, then the statements between the brackets after the do are elaborated.

The first value to be considered for the identifier is that of first expression.

After each elaboration of the statements of the serial clause, a

new value of the identifier is considered. This new value is obtained by adding the value of the second expression to the previous value of the identifier.

To see if the value of the identifier does not exceed the value of the third expression, one has to take into account the sign of the second expression : if it is negative (see the example), 'to exceed' is 'to be less than'.

When the value of the identifier exceeds that of the third expression, the elaboration of the repetitive statement is terminated.

Note that even if the programmer changes a value in E1, E2 or E3 by the elaboration of the serial clause, this has no effect on the control of the loop, because the expressions are calculated only at the beginning of the elaboration of the repetitive clause. This is clear when considering the equivalence given above.

Note : The value of the while -part is calculated and tested before any elaboration of the loop, and thus it can change during that elaboration.

20. Multiple values.

Examples of declarations involving "multiple values".

```
[1:10] real X1, Z1;
[1:10,-10:0] int X2;
[] real Y1 = X1;
[] char FF = "end of file";
[1:80] char BUFFER
```

20.1. Description.

A multiple value is an internal object consisting of

a "descriptor", and a number of values,

the "elements" of the multiple value, each of which

may be selected by "subscripts".

The descriptor may be seen as some number n of "doublets" (Ki,Ui) of integers, i = 1, 2, ..., n.

Ki is the i-th "lower bound",
Ui is the i-th "upper bound".

The number n will be called the "dimension" of the multiple value ; in our implementation, the maximum value of n is 7.

The number of elements of the multiple value is

(U1 - K1 + 1)*(U2 - K2 + 1)*...*(Un - Kn + 1)

if each factor of this product is positive ; otherwise, it is zero.

The subscripts select an element in the usual mathematical way, i.e., to each n-tuple of integers, the i-th of which lies between the corresponding bounds, there corresponds one element of the multiple value.

20.2 Declarations of multiple values.

Examples of identity-declarations involving multiple values were given at the beginning of this section.

Two constructions are possible, with or without an equal-symbol (see 5 and 9). We recall them :

The first one is :

<virtual declarer> <identifier> = <expression>

A "virtual-declarer" for a multiple value is a pair of special brackets, '[' and ']' , the so-called "sub-symbol" and "bus-symbol", with possibly a number of comma-symbols between them, followed by another (virtual) declarer not beginning with brackets.

Examples: [] real
[,,] int
[] ref [,] long int

The number of empty places between brackets and/or commas specifies the dimension of the multiple value. Only the first pair of brackets must be taken into account to determine the dimension.

Note that no bounds are given in a virtual declarer. This is not necessary, because bounds are provided in the right hand side of the declaration.

The modes specified by the declarers above are respectively :

row-of-real	(dimension 1)
row-of-row-of-row-of-integral	(dimension 3)
row-of-reference-to-row-of-long-integral	(dimension 1)

The expression at the right hand side of the equal-symbol must yield a multiple value. Because there are neither denotations nor operations for multiple values in ALGOL68/19 (with the exception of row-of-character, see below), the only possible expression there is an identifier (3d example at the beginning of this section). In ALGOL 68, other expressions are possible.

If the mode is row-of-character, then the expression may be a denotation (see the 4th example above).

The second possible construction is

<actual declarer> <identifier> { , <identifier> }*

See the first two examples at the beginning of this section.

"Actual declarers" for multiple values begin like formal declarers with the fact that "bound pairs" must be given between the brackets and/or the commas, to allow the required memory reservation.

A bound-pair has the form

<expression> : <expression>

: is the "up-to-symbol".

The position of the expressions is strong, they must yield integral values.

Example: [1:N,-1:M over 2] int ABC

Remember that the mode of the identifier ABC is reference-to-row-of-integral.

A declaration of this kind involves the reservation of memory space for a multiple value of the mode specified by the declarer. The value of the first expression of the i-th bound pair becomes the i-th lower bound (see 20.1) of the multiple value, and the value of the second expression of the same bound pair becomes the i-th upper bound.

Another example :

[1:10] ref [],real TRTR

Only the first pair of brackets may have bounds (and must also have bounds in an actual declarer). In this example, memory space is allocated for ten names (addresses).

Note that two pairs of brackets never follow each other, i.e. the elements of a multiple value are never themselves multiple values.

20.3. Use of multiple values.

Examples of statements (taking into account the declarations at the beginning of this section) :

```
 real X ;
X := X1[1];
X1[1]:= 2 ;
X1[2]:= Y1[K+1];
Z1 := X1 ;
X2[1,-1]:= 0
```

The general construction

<identifier>[<expression> { , <expression> }*]

which appears in the examples above is called a "slice", or "subscripted identifier". It is one of the possible syntactic

forms of an expression (see 13).

It selects one element from a multiple value, or its address, according to the rules explained below.

The mode of the identifier must begin with 'row-of', or 'reference-to-row-of'. The latter case may be obtained after dereferencing a number of times. Since the position of the identifier is 'weak', one 'reference-to' must be kept.

The position of the expressions is strong. Since they must yield integral values, only derefencing is possible here. The values of the expressions are the subscripts which select an element of the multiple value (see 20.1).

The number of subscripts must be equal to the number of row-of's at the beginning or after the first reference-to of the mode ; this means that only one element may be selected, and not true 'slices' like in ALGOL 68.

If the mode of the identifier begins with 'row-of', then the mode of the slice is obtained by removing all the row-of's at the beginning of the mode : one element is selected by the subscript(s).

Example : in the assignation

```
X := Y1[1]
```

the mode of Y1 is row-of-real ;
the mode of Y1 [1] is real ;
the selected element is that with subscript 1.

If the mode of the identifier begins with reference-to, then the mode of the slice is obtained by removing all the row-of's which immediately follow the first reference-to ; this means that not an element of the multiple value is selected by the subscript(s), but its address.

This is necessary in assignations like

```
X1[1]:= 3.14
```

The mode of X1 is reference-to-row-of-real ;
The mode of X1[1] is reference-to-real.

Now we see why it is necessary to keep one reference-to in the mode of the identifier.

If the value is needed, and not its address, one dereferencing may occur after the selection of the name by the subscript(s). For example, in

```
X1 [1] := X2[2,-2]
```

the mode of the identifier X2 is

reference-to-row-of-row-of-integral ;

the a priori mode of X2[2,-2]is

reference-to-integral ;

the position of a source being strong, X2 [2,-2] will be dereferenced, then widened to give the a posteriori mode real, which is needed to be assigned to X1[1].

In the assignation Z1 := X1, a multiple value is assigned (after dereferencing) to the name which is the value of Z1.

Another example : FF[2] eq "n" has the value of true .

20.4 The operators upb and lwb .

("upper bound" and "lower bound").

These operators are monadic with priority 10,
or dyadic with priority 8.

As dyadic operators, their first operand is an integer (mode int) their second operand has a mode beginning with 'row of'.

The result is an integral value (mode int), the n-th upper bound (resp. lower bound) of the multiple value of the second operand.

As monadic operators, their operand is a multiple value. The result is the first upper bound (resp. lower bound) of that multiple value.

Examples :
```
X := 3 upb X5 ;
for I from lwb X1 to upb X1 do ....
```

These operators are especially valuable when passing a multiple value to a procedure (see 21) : the bounds, which are part of the value (see 20.1), are passed together with the elements, and can be acceded in the procedure with the aid of the operators upb and lwb .

21. Routines and procedures.

21.1. Routine.

A "routine" is a value (internal object), which may be possessed by a routine denotation or an identifier. The mode of a routine begins with 'procedure'. This explains why we will sometimes write a "procedure" instead of a routine.

A routine is defined as a sequence of symbols which is the same as some block (see 21.3).

In high-level languages, there are generally two constructions of this kind : "subroutines", with or without parameters, which are called by 'call'-statements, and "functions", also with or without parameters, delivering a value, and which can be operands in expressions. ALGOL 68/19 gives the possibility to define only

the former of these two constructions. However, mathematical functions defined in the standard prelude (see 23 and Appendix 3) may be used. In ALGOL 68, both constructions may be defined and used.

A routine may be seen as a piece of program (the routine denotation), which is written at some place, but has to be elaborated at some other place(s) (the call(s)). The parameters are one way to transfer data from or to the calling program. Another way to transfer data is to use the identification conditions (see 27.1) : it is possible to write the program so that an identifier possesses one same value, inside and outside a routine.

The mode of a routine, like that of any other value, is specified by a declarer, the general construction of which is :

proc[(<virtual declarer> { , <virtual declarer> }*)]

Examples :

declarers	mode
proc	procedure
proc (int)	procedure-with-integral-parameter
proc (int , real)	procedure-with-integral-parameter-and-real-parameter

Remember that virtual declarers may not contain bounds, but only brackets and commas, if they contain declarers for multiple values.

Example :

proc ([,] real ,[] int)

Although the modes of the mathematical functions may not explicity appear in any ALGOL 68/19 program, let us mention here that they are particular cases of modes of procedures delivering values. The declarer for such a mode is obtained from the general construction above by adding the mode of the delivered value.

Example :

The mode of the sine function is
'procedure-with-real-parameter-real'
and its declarer could be
proc (real) real

21.2. Declarations.

Examples :

1) proc A = (: skip) # does nothing #

```
2)  proc ( ref real ) B =          # adds 1 to the value #
         ( ( ref real X ):         # to which #
             X := X + 1            # the parameters refers #
         )

3)  proc C = (( real X, ref real Y): Y:=X)
      # assigns to the second parameter the value of the first #

4)  proc D =                                  # permutes the values #
              (( ref  real X , Y) :           # to which the two #
                begin  real Z ;               # parameters refer #
                       Z := X ;
                       X := Y ;
                       Y := Z
                end
              )
```

The general construction for an identity declaration of the first kind (with =) of a procedure-identifier is

```
 proc[( <virtual declarer> { , <virtual declarer> }* )]
<identifier> = ([( <parameter> { , <parameter> }* )]:
                  <statement>
                )
```

where <parameter> stands for

<virtual declarer> <identifier> { , <identifier> }*

The second kind of a declaration is used to declare identifiers of the mode reference-to-procedure... . Examples are :

```
5)  proc E1, E2

6)  proc ( ref  real ) F

7)  proc ( real , ref  real ) G

8)  [1 : 3] proc H
```

The mode of F in (6) is :
reference-to-procedure-with-reference-to-real-parameter.

The mode of H in (8) is :
reference-to-row-of-procedure ; bounds must be given, because the declarer is an actual declarer.

The external objects appearing at the right of '=' in the declarations (1) to (4), and in the general construction following them, are "routine denotations". In ALGOL 68/19, this is the only place where they can appear.

Note that the routine possessed by the identifier (A, B etc.) after the elaboration of its declaration is not the routine denotation, but a slightly altered construction, which will in turn be altered before the elaboration of a call of this routine ; we will only give the final block which is elaborated (see 21.3).

The syntax of a routine-denotation is given above, as the part of the syntax of a procedure declaration at the right of '=' . We note :

- the routine denotation is enclosed between an open- and a close-symbol.

- a "cast-of-symbol" (:) is always present ; it separates the parameters, if any, from the statement ;

- the parameters, if present, are enclosed between an open- and a close-symbol. In this case, there are two open-symbols at the beginning of the denotation ;

- the various kinds of statements are given in the grammar at Appendix 2 (see also 11.1) ;

any one of them, but only one, may be present at the right of the cast-of-symbol. Examples (1), (2) and (3) are clear enough. In example (4), the statement is a block, which is one way to transform more than one statement, together with the necessary declarations preceding them, into a single statement.

Next to the conditions of the grammar of the Appendix 2, the procedure declaration must satisfy the following condition about the parameters :

If the virtual and the formal parameters are present, then for each virtual declarer between the brackets at the left of the equal-symbol, there must be 'something' in the parameter list at the right of that symbol, and in the same sequence :

- either the same declarer followed by an identifier (example 2),

- or an identifier contained in an identifier list following the same declarer (example : proc (int , int) A = ((int B, C) : ...

Otherwise, the virtual declarers group at the left of the equal symbol may be omitted (this will generally be the case).

Example :

proc (int , real) A = ((int B, real C): skip)

may be replaced by

proc A = ((int B, real C): skip)

21.3. Use of routines.

Some examples are given below ; more will be given later. For the following examples, consider the declarations at the beginning of this section, and :

real M,N,X ;

```
 9) E1 := A ;
10) E2 := E1 ;
11) F := B ;
12) G := C ;
13) A ;
14) B(M) ;
15) F(M) ;
16) C(1,M) ;
17) C(M,N) ;
18) D(M,N) ;
19) H[3] := E1 ;
20) M:=SIN(M+1.0)+COS(COS(X));
```

(9) to (12) are assignations (see 12), performed with routines, (13) to (18) are "calls" of procedures delivering no value ; (20) contains three calls of mathematical functions, delivering 'real' values.

After the elaboration of (9), the name (address) possessed by E1 refers to the value (routine) of A. The elaboration of (10) is analogous, but involves first a dereferencing of E1. After the elaboration of (9) and (10), the names possessed by E1 and E2 both refer to the same routine. (11) and (12) show how the modes of the source and of the destination must correspond in an assignation.

The general construction of a call is

<identifier>[(<expression> { , <expression> }*)]

The positions of the identifier and of the expression are strong (see 17).

Note : H[3] is not a call, because no subscript is allowed in a call. To call the intended procedure, two instructions have to be used, for example :

```
E1 := H[3] ; E1;
```

(remember that E1 has the mode 'procedure').

The mode of the identifier - after dereferencing if necessary, must begin with procedure. The mode of each expression - after coercion(s) if necessary - must be the same as that of the corresponding parameter in the declaration of the identifier (see hereafter).

The elaboration of a call of a procedure delivering no value is the elaboration of the block obtained as follows :

1. A copy is made of the routine-denotation of the declaration of the identifier of the call.

2. Each parameter is replaced by an identity-declaration, where the left hand side of the equal-symbol is the parameter itself, and the right hand side the corresponding expression in the call (the number of expressions must of course be equal to the number of identifiers in the declarations, with appropriate modes).

This would give, with (2) and (14) above :

ref real X = M

With X instead of M in the call, we would however come to

ref real X = X

which is unacceptable. To avoid such a situation when it appears, each occurrence of X inside the declaration of B should be replaced by another identifier, not already contained in the program ; this replacement must occur before substituting the declarations for the parameters. In the last example, one could obtain :

ref real X1 = X.

3. The open-symbol of the parameter list is deleted.

4. The close-symbol of the parameter list and the cast-of-symbol following it are replaced by a goon-symbol (;).

5. All comma-symbols of the parameter list are replaced by goon-symbols.

6. The open-symbol and the close-symbol of the routine-denotation are respectively replaced by a begin-symbol and a end-symbol.

The copy thus modified is elaborated ; if in the routine-denotation there is no jump (see 11.4) to a label outside that block, the next statement will be the statement following the call, if any.

Examples :

1. The elaboration of A is the elaboration of :

```
begin skip end
```

2. The elaboration of B(M) is that of

```
begin ref real X = M ;
      X := X + 1
end
```

3. The elaboration of B(X) is that of

```
begin ref real X1 = X ;
      X1 := X1 + 1
end
```

The call B(3.14) is incorrect, because 3.14 has the mode real , whereas the parameter has been declared ref real , and there is no mode conversion adding a ref before another mode. Such a call would be rejected by the syntactic analyzer.

On the other hand, C(M,N) and C(1,N) are correct ((16) and (17)) because the position of the expressions is strong : the M will be dereferenced to have the mode real, whereas the 1 will be widened to the mode real.

The position of the identifier in the call is also strong : a

dereferencing of F will take place before the elaboration of the call F(M) of (15).

The elaboration of a call of a procedure delivering a value follows the same rules as above, with the exception that the block has a value, which is the value of the call in the expression in which it is contained.

For example, the value of the call COS(X) is the 'real' value which is the nearest to the cosine of the value possessed by X.

21.4. Separately compiled procedures.

Programs using "separately compiled procedures" must be preceded by pseudo-declarations to specify the mode of these procedures ; the pseudo-declarations are enclosed between two pr 's (pragmat-symbols), and separated by comma-symbols.

The separately compiled procedures themselves are simply procedure-declarations.

The order of compilation of separately compiled programs has no special meaning.

Example of a separately compiled procedure, itself using another separately compiled procedure.

```
pr proc(int) PROG1 pr

begin
     .......
     PROG1(3)
     .......
end % of the main program %


pr  proc PROG2 pr
proc PROG1 =
      (( int I) :
         if I gt 0
         then PROG2
         fi
      ) # end of first procedure #


proc PROG2 = (: PUTS("....") )
```

Note that absolute security is given to the programmer with the use of pragmats and separately compiled procedures:

- if, in a pragmat, the mode of an identifier doesn't correspond with the mode of an external procedure or with the mode of a previously compiled pragmat, a syntactical error arises.

- a test is provided to verify if the mode of a separately

compiled routine corresponds to some pragmat (if no pragmat is found, one is created).

- no test on pragmats occurs if option NODECK is active (this has no consequence on security, because no TEXT file is generated).

Note that the compiler uses and saves CMS files with FILETYPE 'PRAGMAT' to compile such programs.

If an error concerning a pragmat-identifier is detected, all TEXT files of programs using this pragmat-identifier are erased. Furthemore, the 'identifier PRAGMAT' file is also erased.

21.5. "Common" identifiers.

For further information about this feature, see Appendix 6.

21.6. Passing labels as parameters.

A trick can be used to pass a label as parameter or to have a label in common. This is achieved by using a procedure:

```
proc GOTO1 = (: goto LABEL1 )
```

You pass GOTO1 as parameter to another procedure or you make GOTO1 an identifier in common. Calling this parameter will have the desired effect.

22. Deleted.

23. Prelude and postlude.

A program actually written by a programmer in ALGOL 68 (called a "particular program") must be seen as beeing enclosed in an outer block, containing some declarations. This block contains the so-called "standard prelude" and "library prelude", preceding the program, and the "standard postlude", following it.

The standard prelude of ALGOL 68/19 contains the operator-declarations (see 15) (there are no explicit operator declarations in ALGOL 68/19) and the declaration of the mathematical functions (see 24).

The library prelude contains all implicit declarations, own to a particular implementation. The library prelude of ALGOL 68/19 is given in Appendix 3. It contains the declarations of input/output (see 25) and conversion routines (see 26), and other utility routines.

Separately compiled procedures, when being called in another program, must also be thought as being contained in the library

prelude.

The standard postlude contains :

```
EXIT: CP("CLOSE RDR"); CP("CLOSE PUN");
      CP("CLOSE PRT NAME RUNLIST ALGOL")
```

Note that every program goes through this standard postlude, even if it terminates abnormally (with runtime errors).

24. Mathematical functions of the standard prelude.

"Mathematical functions" are provided in ALGOL 68/19 under the form of procedures delivering values, to be used in calls.

Example :

```
F := SIN(X) ;
```

The mathematical sinus of the value of X is computed, and the result is the value of the call SIN(X) ; its mode is 'real'.

All mathematical functions of the library prelude are built along the same line ; their names suggest the computed function, together with the length of the argument and result. The numerical algorithms are those used in the corresponding FORTRAN-functions from the IBM/360 DOS system.

The table of Appendix 3 (A3.3) gives the identifiers used, together with their modes. Restrictions on the values of the parameter are also given, and the corresponding error code when the calculation fails.

Examples of the use of mathematical functions.

1. To calculate the sum of the squares of a sine and a cosine.

```
real X,Y ; X :=0.39 ;
Y:= SIN(X)**2+COS(X)**2 ;
```

2. To calculate a non integral power of a real number.

```
real A,X,Y ;
A:= 3.14 ; X:= 3.15 ;
Y:= EXP(X*LN(A)) ;
```

Note : A**X is not permitted because X is real, but the routines (L)EXP and (L)LN may be used for the purpose.

25. Input and output routines.

A complete set of "input/output routines" is provided in ALGOL 68/19. They are not those of ALGOL 68.

Access to, and control of:
one virtual card reader,
one virtual card puncher,
one virtual line printer,
one virtual console,
virtual tapes and disks.

is made possible by those routines. They are basic in the sense that they achieve the basic transfers of data (characters or bytes) and control of equipement. More elaborate routines can be written by the user, but all basic possibilities of the equipement are available.

Blocking and deblocking of the records must be done by the user. No simultaneity of input, output and processing is made possible.

Conversion of data from and to character string can be done by using another set of routines (see 26).

The examples at the end of section 26 show how the input and output routines should be used in connection with the conversion routines.

26. Conversion routines. Input-output with conversion.

26.1. Basic conversion routines.

The "conversion routines" of Appendix 3 (A3.3) convert data to or from character string representation, from or to internal code corresponding to the mode of the data. The character string considered is part of another character string (in a typical use, an output or an input buffer).

They will normally be used in conjunction with the input and output routines mentioned in section 25. They are given by pairs, one normally used on input, the other one on output, for the same mode of data.

Examples.

1. Reading a card containing a character string and a real number.

```
[1:80] char B; [1:10] char TITLE ; int PAGE;
 READ(B,1,80);
 FRC(B,1,TITLE,1,10) ;
 INI(B,21,2,PAGE);
```

2. Writing a line with a title and a page number (consider also the declarations of example 1 in connection with the following instructions).

```
[1:120] char LINE;
  BLANK(LINE) ;
  FILL(LINE, 110, 4, "PAGE");
  TRC(TITLE,1,LINE, 30, 10);
  OUTI(LINE, 115, 2, PAGE);
  WRITE(1,LINE, 1, 120);
```

26.2. Input-output of one element with conversion.

A simplified set of combined input or output, and conversion routines is also part of the library prelude (see A3.6). They make it possible to read one elementary value directly into a 'variable', and to write such a value on a line of the printer.

Example :

```
 int A,B,C,D;
GETI(A); GETI(B); GETI(C); GETI(D);
PUTR((A+B+C+D)/4);
```

These instructions read 4 integral numbers, each one being punched on a separate card, and print their arithmetic mean on one line of the printer with a standard floating point format.

26.3. Formatted input-output.

"Formatted input-output" is made possible using three routines : FORMAT, GET, PUT.

The first one activates a "format" string to be used in the future input and output calls of "GET" and "PUT". The routines GET and PUT have a variable number of parameters. Their elaboration is described at appendix 10, together with their associated format calls.

Simple example :

```
int M,N ; M:=3 ; N:=4 ;
FORMAT(3,"2(I4)") ;
PUT(M,N);
FORMAT(1,"STREAM"); [1:10]real T; GET(N,T,M);
```

27. Identification. Context. Scope.

27.1. The identification process.

A "range" is either a block, or a routine denotation or a do statement.

Each occurrence of an identifier in a program is either a "defining occurrence", or an "applied occurrence".

A given occurrence of an identifier is a defining occurrence if it follows a declarer, or if it is a label (see 11).

Otherwise it is an applied occurrence.

A given applied occurrence of an identifier may "identify" some defining occurrence of the same identifier by the following steps :

Step 1 : The given occurrence is termed the 'home', and step 2 is taken ;

Step 2 : If there exists a smallest range containing the home, then this range, with the exclusion of all ranges contained within it, is termed the home, and step 3 is taken ; otherwise, there is no defining occurrence which the given occurrence identifies.

Step 3 : If the home contains a defining occurrence of the considered identifier, then the given occurrence identifies it ; otherwise, step 2 is taken.

Example : in the following example, a defining occurrence of an identifier and an applied occurrence of the same identifier are placed approximatively along the same vertical line.

```
begin  real A ;  begin  real A ;
                          A := 1
                 end ;
           A := 1
end
```

27.2. The identification conditions.

(1) The mode of a defining occurrence of an identifier, and that of an applied occurrence of the same identifier which identifies the first one, are one same mode.

This means that the mode specified in a declaration for a given identifier must be considered as the definition of the mode of the same identifier in all applied occurrences which identify the first one.

(2) Each applied occurrence of an identifier must identify some defining occurrence of the same identifier (there must be a declaration for every identifier or it must appear somewhere as a label ; the condition is in fact stronger, see 27.1).

It should be noted that the defining occurrence of some identifiers can be found in the standard prelude (see 23), or in the library prelude of the particular implementation ; this is the case for the label EXIT , and for the identifier SIN, for example, in our implementation (see 24 to 26). Further, a true declaration may be replaced by a pseudo-declaration at the head of the program, between two pr-symbols (see 21.4.) or two co-symbols (see 21.5.).

One additional rule must be respected in ALGOL 68/19 : a declaration

of an identifier must (lexicographically) precede each applied occurence of it (this rule does not apply to labels).

As a consequence, to declare two mutually recursive procedures, the following trick has to be used :

```
proc P;
proc A = (:... ;P; ...);
proc B = (:...;A; ...);
P:=B;
```

Another rule, which has nothing to do with identification, may be mentioned here : before elaborating an instruction with an applied occurrence of an identifier, the declaration containing the defining occurrence of that identifier must have been elaborated.
This makes it possible to write meaningful programs like this :

```
int N;
GETI(N);
[1:N] real X;
```

However, the effect of these programs is undefined (see A7.):

```
  ...                ...              pr proc A pr
goto L;            goto L;              ....
int I;             proc A = ...         A; %assume A uses I%
L: I:=1;         L: A;                  int I;
  ...                ...              ....
                                      SAVE COM(I) %see 21.5.%
```

27.3. The uniqueness condition.

A "reach" is a range, with the exclusion of all its constituent ranges.

(3) Two defining occurrences of the same identifier may not appear in the same reach.

Example : `real X ; real X ; int X ;` is not allowed, but the example of 27.1 above is correct.

27.4. Context conditions.

The conditions (1), (2) and (3) are called "context conditions". They are checked at compile time.

27.5. Scope conditions.

We are dealing here with a problem of another kind. It is given together with the context conditions, because it has something to do with ranges.

Each value (this is an internal object) has one specific "scope".

The scope of a plain value is the program (see 4 and 11).

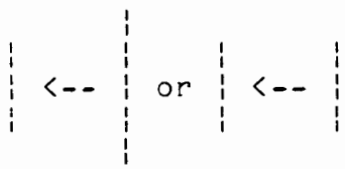
The scope of a name is the smallest range containing the declaration of the identifier which possesses that name.

The scope of a multiple value, and that of a routine, will be given below.

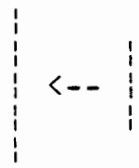
Let us first see which problems arise in connection with scopes. It must first be understood that there are only problems with values involving names. There is a "scope conditions" in assignations :

(4) The scope of the source must be not smaller than the scope of the destination.

This may be sketched as follows (the brackets represent scopes of corresponding values in an assignation from right to left) :



is allowed.



is not allowed.

Examples of programs :

```
begin
   int Y ;
  Y := 2 ;
   begin
     ref  int YY ;
    YY := Y
   end
end
```

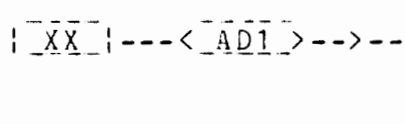
is allowed.

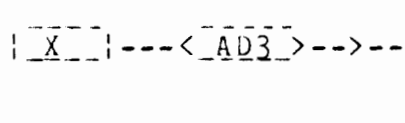
```
begin
  ref  int XX ;
   int Y ;
  begin
   int X ;
    X := 1 ;
     XX := X
  end ;
 Y := XX
end
```

is not allowed.

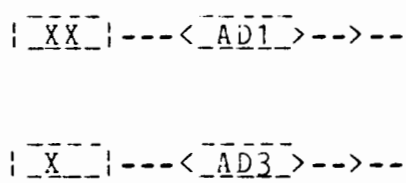
The reason why the construction on the right is not allowed may be explained by the following sketches :

') State after elaboration of ref int XX (outer block) and int X (inner block) followed by X := 1 :

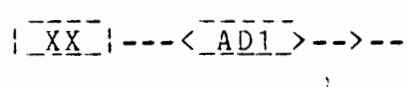




2) State after elaboration of XX := X : the address (1) has been copied into (2) :



3) After leaving the inner block, the right half of the sketch above 'disappears' i.e. the corresponding memory space may have been freed and used for other purposes :



The address (3) now refers to an unpredictable value (perhaps to no value at all). The assignation Y := XX is no longer possible.

The scope of a multiple value is the smallest of the scopes of its elements, if any ; otherwise, it is the program. The scope of a multiple value can change during elaboration.

Example :

```
begin
    [1:2] ref real YY ;
      real X1 ;
      begin  real X2 ;
             begin [1:2]  ref real XX ;
                    XX[1] := X1 ;
                    XX[2] := X2 ; #1#
                    XX[2] := X1  #2#
             end
      end
end
```

The name XX refers to a multiple value, the elements of which are names.

After the elaboration of the assignation #1#, the scope of that multiple value is that of X2 ; after the elaboration of #2#, it is that of X1.

It is necessary to know the scope of a multiple value, before assigning it to a name. The assignation YY := XX would not be valid, if it took the place of the assignation #2#, but it would after it.

Being a dynamic condition, the scope condition cannot be checked at compile time. In our implementation, it is not checked at elaboration time either. When not satisfied, it can produce unpredictable results.

The scope of a routine possessed by a given denotation D is the smallest range containing that denotation.

Example with procedures:

```
begin
    proc PROC1 = ((ref proc PAR):
      begin
        proc A = (: ..... );
        if # a boolean EXPR # then PAR:=A else PAR:=STOP fi
      end
    );
    proc VAR;
    PROC1(VAR);
    VAR  # this program violates the scopes condition
           if the value of EXPR is true #
end
```

A1. NON EXISTING

A2. A context-free grammar of ALGOL 68/19.

The writing rules of the "context-free grammar" use the following meta-symbols :

< and > to enclose "meta-notions" ;

::= to separate a meta-notion from its definition rules (this meta-symbol can be read : 'is a', or 'must be a') ;

¦ (read : 'or') to separate alternate definitions ;

{ and } to enclose two or more alternate definitions ;

{ and }* to enclose a part of a definition rule which may be reproduced a number of times, including 0 ;

{ and }+ idem, except : at least one time ;

{ and }0 idem, except : 0 or one time, i.e. optional part.([and] are also used in the manual itself; when some ambiguity arises, refer to this appendix).

Other marks stand for themselves, and are ALGOL 68 symbols. Meta-notions and/or ALGOL 68 symbols, when not separated by meta-symbols, must follow one another (but see 7).

Example : the rule for <decl> should be read (somewhat expanded) :

A declaration is

either an actual declarer followed by an identifier, and possibly followed by other identifiers, with commas as separators, or a virtual declarer followed by an identifier, followed by an equal-symbol, followed by either an expression or a routine.

There may be some slight departures from the rules given above, but we think they are obvious enough to need no further explanation.

1.<program>::={ pr <virt decl suite> pr ¦ co #see 21.5.# co }*
{ {<label>:}0 <block> { SAVE #see 21.5# }0 ¦
<virt decl> <id> = <routine> }

<block>::= begin <serial clause> end

<serial clause>::={{<stat>;}*<decl>;}*
{<label>:}0<stat>{;{<label>:}0<stat>}*

<routine>::=({(<param> {,<param>}*) }0: <stat>)

<param>::=<virt decl><id> {,<id>}*

2.<decl>::={<act decl><id> {,<id>}* ¦
<virt decl><id> = {<expr> ¦ <routine> }}
<virt decl suite>::=<virt decl><id>{,<id>}*{,<virt decl><id>{,<id>}*}*

```
<virt decl>::={[{,}*]}0 <virt nonrow decl>

<act decl>::={ [<bound>{, <bound>}*] }0<virt nonrow decl>

<bound>::=<expr> : <expr>

<virt nonrow decl>::= { ref <virt decl> |
          { long }0{ int | real }     | bool | char |
            proc { (<virt decl> {,<virt decl>}* ) }0       }

3.<stat>::={ goto <id> | skip | <block> | <expr> := <expr> |
          <id> { (<expr> {,<expr>}* ) }0 |
           if <expr> then <stat> {;<stat>}*{ else <stat>{;<stat>}*}0 fi |
          { for <id>}0{ from <expr>}0{ by <expr>}0{ to <expr>}0
          { while <expr>}0 do ( <serial clause> )   |
          case <expr> in <stat>{,<stat>}*{ out <stat>}0 esac }

4.<expr>::=01

  <0i>::= {<0i+1> | <0i> <0pi> <0i+1> }    ,   i from 1 to 9
  <010>::={<0p10><010> | (<expr>) | <base> }

  <base>::={<id> {[ <expr> {, <expr>}*]}0 | <denotation> |
          <id>(<expr>{,<expr>}*)}

  <0p2>::= or    <0p3>::= and    <0p4>::= { eq | ne }
  <0p5>::={ lt | le | ge | gt }      <0p6>::= { - | + }
  <0p7>::={ * | / | over | mod }     <0p8>::= { ** | lwb | upb }
  <0p10>::={ round | sign | odd | snort | leng | repr | abs |
             not | upb | lwb | entier | + | - }

5.<id>::=<label>::= <letter> {<letter>|<digit>}*

  <letter>::={ A | ... | Z }     <digit>::={0 | ... | 9 }

  <denotation>::= {{long}0{<digit>}+|  true | false |"{<item>}*" |
                  { long }0{<digit>}+ E {+|-}0{<digit>}+ |
                  { long }0{<digit>}*.{<digit>}+{E{+|}0 {<digit>}+}0}

  <item>::= { <any character, except the quote-character (' or $)> |
              <double quote-character ('" or $$)> }
```

A3. Preludes and libraries.

A3.1. Routines of the library prelude.

The 'ALG68LIB TXTLIB S2' CMS file contains the library prelude of ALGOL68/19.

Two kinds of routines can be found in this library :

- "standard routines"

  mathematical routines : (see A3.2)

  EXP, LN, SQRT, SIN, COS, TAN, ARSIN, ARCOS, ARTAN,
  LEXP, LLN, LSQRT, LSIN, LCOS, LTAN, LARSIN, LARCOS, LARTAN.

  conversion routines : (see A3.3)

  TRC, FILL, INI, OUTI, INLI, OUTLI, INR, INLR, OUTR, OUTLR,
  OUTFR, OUTFLR, INB, OUTB, COMP.

  input-output routines : (see A3.4)

  READ, WRITE, LINE, PAGE, PUNCH, ACCEPT, DISPLY, TAPER, TAPEW,
  WTM, REW, BSR, CLOSE, DISKR, DISKW, FORMAT, GET, PUT.

- "non-standard routines"

  which contain among others simplified input-output routines (see A3.6) :

  TAKE, STOP, EOF, GETI, GETLI, GETR, GETLR, GETB, GETS,
  PUTI, PUTLI, PUTR, PUTLR, PUTB, PUTS, TIME, DATE, BLANK,
  DUMP, CMS, CP, ON, RESET.

Note: Programmer routines, which are the separately compiled procedures (see 21.4) and which need "pragmats" to be used may be catalogued in some 'TXTLIB' library (see A5.4).

A3.2. Standard routines : Mathematical functions.

identifier	mode	error code	range of parameter A
EXP	proc (real) real	50	A le X'42AEAC4F' (1)
LN	"	51	A gt 0
SQRT	"	52	A ge 0
SIN	"	53	abs A lt X'45C90FDB'
COS	"	54	"
TAN	"	55	" (2)
ARSIN	"	56	abs A le 1
ARCOS	"	57	"
ARTAN	"	58	no limits
LEXP	proc (long real) long real	59	A le X'42AEAC4F'
LLN	"	60	A gt 0
LSQRT	"	61	A ge 0
LSIN	"	62	abs A lt X'4DC90FDA'
LCOS	"	63	"
LTAN	"	64	" (2)
LARSIN	"	65	abs A le 1
LARCOS	"	66	abs A le 1
LARTAN	"	68	no limits

(1) This denotation means 'hexadecimal 42AEAC4F'

(2) Argument not to close to a (2k+1)-multiple of PI/2.

A3.3. Standard routines: Conversions.

A3.3.1. Characters.

There is no conversion here, but only transfer of "substrings".

a. Transfer into the second string of a substring of the value referred to by the first parameter.

```
proc TRC =
        (( ref [] char A,   # First string. #
           int I,           # Starting at the character A[I], #
           ref [] char B,   # transfer into B, #
           int K,           # starting at the character B[K], #
           int J):          # a substring of length J#
        )
```

Error 11, if one of the following conditions is not satisfied :

```
I  ge  lwb A                          K  ge  lwb B
I+J-1  le  upb A                      K+J-1  le  upb B
                    0 lt J
```

b. Transfer into the first string the value of the second one.

```
proc FILL =
        (( ref [] char A,  # first string. #
           int I,          # A field beginning at the character A[I] #
           int J,          # and with length J, will receive the #
           []  char X):    # string X. If X is too long, it #
                           # will be cut at the right ; if too short #
                           # it will be left adjusted in the field #
                           # and padded to the right with blanks #
        )
```

Error 12, if one of the following conditions is not satisfied :

```
I  ge  lwb A
I+J-1  le  upb A
0 lt J
```

Note : use BLANK to fill a string with blanks.

c. Comparison of substrings.

```
proc COMP =
  (([]char A, int I, []char B, int J,K, ref int R):
     #
        Comparison of two substrings, the first beginning at A[I],
        the second at B[J], with length K; the value of the
        result R is:
           -1 , when the first substring is less than the second,
            0 , when the first substring is equal to the second,
           +1 , when the first substring is greater than the second.
        The comparison is made using the 'CLCL' ASSEMBLER instruction,
        which performs a logical comparison between the 'hexadecimal
        values' representing the strings.

        Error 10 occurs when:
              K le 0,
              I lt lwb A,
              J lt lwb B,
              I+K-1 gt upb A,
              J+K-1 gt upb B.
     #
  )
```

A3.3.2. Integers.

a. Conversion of characters of a string to an integral value.

```
proc INI =
        (( ref [] char A, # String A contains the string #
                          # to be converted. #
           int I          # A field beginning at A[I], #
           int J,         #    and with length J if J gt 0, #
                          #        with length 2 if J le 0, #
           ref int Y):    # will become the value to which Y refers, #
                          #   with conversion (see below) if J  gt 0, #
                          #   without conversion otherwise #
        )
```

Error 15 if one of the following conditions is not satisfied :

- if J gt 0 : I ge lwb A (conversion)
  I+J-1 le upb A
  J le 256
  the field contains at least one digit, following an optional + or - sign, with optional blanks at any place. No other character is used.
  leading zeroes are ignored.
  the corresponding integral value is less than 32767 and greater than -32768.

- if J le 0 : I ge lwb A (no conversion)
  I+1 le upb A

b. Conversion of an integral value to characters.

```
proc OUTI =
        (( ref [] char A,  # String A receives the converted #
                           # value. #
           int I,          # A field beginning at the character #
           int J,          # A[I], and with length J if J gt 0, #
                           #          with length 2 if J le 0, #
           int X):         # will receive the value of X, #
                           # converted to decimal characters if #
                           # J  gt 0,#
                           # without conversion otherwise. #
        )
```

The converted value is right adjusted, and has a minus-sign if necessary.

Error 17 if one of the following conditions is not satisfied :

- if J gt 0 : I ge lwb A (conversion)
  I+J-1 le upb A
  J le 255
  the field is long enough to receive the value

- if J le 0 : I ge lwb A (no conversion)
  I+1 le upb A

c. Conversion of characters of a string to a long-integral value.

```
proc ( ref [] char , int , int , ref long int ) INLI
      =# see INI, except that if J le 0, #
       # then the length of the field is 4 instead of 2 #
```

Error 16 in the same conditions as for error 15, except that :
the limits of the integral value are 2**31-1 and -2**31 ;
if J le 0, then I+3 le upb A.

d. Conversion of a long-integral value to characters.

```
proc ( ref [] char , int , int , long int )
  OUTLI  # see OUTI, except that if J le 0, #
         # then the length of the field is 4 instead of 2 #
```

Error 18 in the same conditions as for 17 in OUTI, except that
if J le 0, then I+3 le upb A

A3.3.3. Real numbers.

There are two input routines, one for the mode 'real', and one for the mode 'long-real', and four output routines, one for every length number, one for output with or without exponent part.

a. Conversion of characters of a string to a real value.

```
proc INR =
        (( ref [] char A,  # String A contains the string #
                           # to be converted. #
           int I,          # A field beginning at A[I], #
           int J,          #      and with length J if J  gt 0 #
                           #          with length 4 if J  le 0 #
           ref  real Y):   # will become the value to which Y refers,#
                           # with conversion (see below) if J  gt 0, #
                           # without conversion otherwise. #
        )
```

Error 19 if one of the following is not satisfied :

- if J gt 0 : I ge lwb A (conversion)
  I+J-1 le upb A
  J le 256
  the field contains a number in decimal form, consisting of
  - either an optional + or - sign,
    an integral part of 1 to 9 digits,
    the letter E followed by an optional + or - sign and
    an exponent of 1 or 2 digits,
  - or an optional + or - sign,
    an optional integral part of m (le 9) digits,
    a point,
    a fractional part 1 to 9-m digits,
    an optional (letter E, followed by
    an optional + or -sign, and
    an exponent of 1 or 2 digits),

with blanks and non significant zeroes used freely throughout the field.

The value of the number, when normalized under decimal form, must have an exponent not greater than 75 in absolute value.

Examples of valid strings : 18 E 25 -55-3
.1 2.5E3
1. and 1.E3 are not valid.

- if J le 0 : I ge lwb A (no conversion)
I+3 le upb A
the value must be a normalized floating point number.

b. Conversion of characters of a string to a long-real value.

```
proc ( ref [] char , int , int , ref long real )
  INLR =# see INR, except that for J le 0, the length of the #
        # field is 8 #
```

Error 20 in the same conditions as for 19 (INR), except that the number of significant digits may be up to 18, and if J le 0, then one condition is I+7 le upb A.

c. Conversion of a real value to characters, with exponent part.

```
proc OUTR =
        (( ref [] char A,# String A receives the #
                         # converted value. #
           int I,        # A field beginning at the character #
           int J,        # A[I] , and with length J if J gt 0 #
                         #            with length 4 if J le 0 #
           int K,        # will receive with K significant decimal #
                         # digits if J gt 0, without conversion#
           real X):      # otherwise, the value of X #

        )
```

The format of the converted value is a right-adjusted number, padded to the left with blanks, and containing a - sign if necessary,
a point,
K decimal digits,
the letter E,
a + or - sign,
an exponent of 2 digits.

Normal rounding takes place, if necessary.

Error 21 will be raised if one of the following conditions is not satisfied :

- if J gt 0 : I ge lwb A (conversion)
I+J-1 le upb A
J le 256
J ge K+6
K gt 0

```
                K  lt 19
- if J  le 0 :  I  ge  lwb A          (no conversion)
                I + 3  le  upb A
```

d. Conversion of a long-real value to characters, with exponent part.

```
proc ( ref[]char, int, int, int, long real ) OUTLR
        =# see OUTR, except that the length of the field is 8, #
         # if   J  lt 0 #
```

Error 23 in the same conditions as error 21 (OUTR), except:

```
  if J  le 0 :  I + 7  le  upb A
```

e. Conversion of a real value to characters, without exponent part.

```
proc OUTFR
          (( ref [] char A, # like OUTR #
             int I,         # but without exponent part. #
             int J,         # If J  le 0, no difference with OUTR #
             int K,
             real X):
          )
```

The format is a right-adjusted number, padded to the left with blanks, and consisting of : a - sign if necessary,
an integral part of S digits, depending on the value,
a point,
a fractional part of K digits.

Error 22 will be raised if any of the following conditions is not satisfied :

```
 -if J  gt 0 : I  ge  lwb A          (conversion)
               I+J-1  le  upb A
               J  le 256
               J  ge K+S+2
               K  gt 0
               K + S   le 18

 -if J  le 0 : I  ge  lwb            (no conversion)
               I+3  le  upb A
```

f. Conversion of a long-real value to characters, without exponent part.

```
proc ( ref[]char, int, int, int, long real ) OUTFLR
        =# see OUTFR, except that the length of the field is 8 for #
         #  J  lt 0 #
```

Error 24 in the same conditions as error 22, except,

if J le 0: I+7 le upb A

A3.3.4. Boolean values.

a. Conversion of characters of a string to a boolean value.

```
proc INB =
        (( ref [] char A,   # String A contains the string #
                            # to be converted. #
           int I,           # A field beginning at A[I], #
           int J,           # and with length J, if J gt 0, #
                            #     with length 1, if J le 0, #
           ref  bool Y):    # will be scanned, to find a value     #
                            # to be assigned to Y (see below) #
        )
```

if J le 0, then there is no conversion ;

if J gt 0, then the field is scanned, blanks are ignored,
if the first non blank character is '1', then true is
assigned to Y,
if it is '0', then false is assigned to Y.

Error 13 if one of the following conditions is not satisfied :

- if J le 0 : I ge lwb A (no conversion)
  I le upb A
- if J gt 0 : I ge lwb A (conversion)
  I+J-1 le upb A
  J le 256
  the field contains more than one non blank character,
  or a non-0 or non-1 character.

b. Conversion of a boolean value to characters.

```
proc OUTB =
        (( ref [] char A,   # String A receives the #
                            # converted value. #
           int I,           # A field beginning at A[I], #
           int J,           # and with length J if J  gt 0, #
                            #     with length 1 if J  le 0,#
           bool X):         # will receive the value of X #
                            # (see below) #
        )
```

if J le 0, then there is no conversion ;

if J gt 0, then the field receives a '1' in its rightmost
position if the value of X is true , and a '0' otherwise.
The other positions of the field, if any, are filled with blanks.

Error 14 if one of the following conditions is not satisfied :

```
- if J gt 0 : I ge lwb A          (conversion)
              I+J-1 le upb A
              J le 256

- if J le 0 : I ge lwb A          (no conversion)
              I le upb A
              I+J-1  le  upb A              I  le  upb A
              J  le 256
```

A3.4. Input and output routines.

A3.4.1. Output to the printer.

```
proc WRITE =
        ((   int K,              # Skip K lines before printing. #
             ref []  char A,     # String A contains #
                                 # the data to be output. #
             int I,              # Starting from A[I], #
             int J) :            # output J characters #
                                 # to the first J places of the #
                                 # current line. #
        )

proc LINE =
        ((  int K ) :        # K lines are skipped immediately. # )

proc PAGE = (: #body #)      # The printer is positioned at the #
                             # first line of the next page #
```

Limits : 0 le K lt 3
0 lt J le 132
lwb A le I
I+J-1 le upb A.

If not respected, error 30, for the three routines.

An error 30 also occurs, when something wrong arises with the virtual printer (not ready, not attached,.....).

A3.4.2. Card reading and punching.

```
proc READ =
       ((  ref []  char A,# String A will contain #
                          # the data after input. #
           int I,         # Start filling A from A[I], #
           int J):        # with the first J characters of the #
                          # next record  #
       )
```

```
proc PUNCH =
        ((  ref []  char A,# String A contains #
                           # the data to be output. #
            int I,         # Starting from A[I] #
            int J) :       # output J characters into the #
                           # first J positions of #
                           # a new record   #
        )
```

Notes : Limits : lwb A le I
I + J - 1 le upb A
0 lt J le 80 for READ and 0 lt J le 80 for PUNCH

Errors if not respected : 31 (READ) or 32 (PUNCH).

An "end of file" is not detected by READ, but see TAKE.

An error 31 (READ) or 32 (PUNCH) also occurs, when something wrong arises with the virtual RDR or PCH (not ready, not attached,....).

A3.4.3 Access to the terminal.

```
proc ACCEPT =
        ((  ref []  char A,# String A will contain #
                           # the data after input. #
            int I,         # Start filling A from A[I] #
            int J):        # with J characters #
                           # entered from terminal. #
        )

proc DISPLY =
        ((  ref []  char A,# String A contains #
                           # the data to be output. #
            int I,         # Start position in A : A[I]; #
            int J):        # output J characters into the #
                           # first J positions of a new line #
                           # of terminal. #
        )
```

Notes : Limits : see card reading and punching, except 0 lt J le 130
Errors if not respected : 33 (ACCEPT) or 34 (DISPLY)
An error 33 (ACCEPT) or 34 (DISPLY) also occurs when something wrong arises with the virtual console (not ready, not attached, ...).

A logical "end of file" ('*EOF' characters) is detected by ACCEPT and provocates an error 38.

A3.4.4 Access to magnetic tapes.

```
proc TAPER =
        ((  int K,              # CMS number of the virtual tape,
                                    1 (181) , 2 (182) , ...   #
            ref [] char A,      # String A will contain #
                                # the data after input. #
            int I,              # Start filling A from A[I], #
            int J,              # with the first J characters #
                                # of the next record of the tape. #
            proc P):            # P is called on end of file. #
        )

proc TAPEW =
        ((  int K,              # CMS number of the tape #
            ref [] char A,      # String A contains #
                                # the data to be output. #
            int I,              # Starting at A[I]     , #
            int J):             # J characters will be used to#
                                # write a record on tape. #
        )


proc WTM =
        ((  int K) :                # CMS number of the tape #
                                    # write a tape-mark. #
        )

proc REW =
        (( int K):                  # CMS number of the tape #
                                    # rewind the tape. #
        )


proc BSR =
        (( int K):                  # CMS number of the tape #
                                    # back-space one record. #
        )
```

Limits : lwb A le I I + J - 1 le upb A
1 le K le 4 J gt 0 in TAPER
J gt 17 in TAPEW

Error 35 if one of the conditions is not satisfied.
An error 35 is also detected, when something wrong arises with the tape (unit not attached, tape not ready,...).

Notes :

1. Alternate tape-switching is ignored.

2. When, on input, J is greater than the physical length of the record, filling with blanks occurs at the right.

3. One of the following actions can be the cause of a tape input-output error later on:
   - "over-writing" a record (not when extending an existing file by searching for the tape-mark, backspacing one record and starting to write) ;

- a call of TAPER , followed by a call of TAPEW or WTM.

A3.4.5. Access to magnetic disks.

```
proc CLOSE = (([]char FILE):
   # closes the current CMS-FILE # )

proc DISKR =
  (([]char FILE, int I, ref[]char A, int J, K, proc EOF):
       # reads a record from the current CMS-FILE;
         I is the position of the record in the FILE,
         string A will contain the data after input;
         filling A begins at A[J], with the K first characters
         of the I-th record if I gt 0, or of the next record
         if I le 0.
         EOF is called when the end of the file is reached.
       #
  )

proc DISKW =
  (( [] char FILE, int I, ref [] char A, int J, K ):
        # writes a record on the current CMS-FILE;
          I is the position of the record in the FILE;
          string A contains the data to be output;
          starting at A[J], K characters will be used
          to construct the I-th record if I gt 0, or
          the next record if I le 0
        #
  )
```

Error 37 occurs if one of the following conditions is not satisfied in DISKR or DISKW:

```
   J ge lwb A    J+K-1 le upb A
   0 le I        1 le K le physical record length
```

Error 36 (CLOSE) or 37 (DISKR or DISKW) also occurs when something wrong arises with the FILE (file not found,...) (see the ASSEMBLER macros FSOPEN, FSCLOSE, FSREAD, FSWRITE).

Notes:

- the identifier FILE is the CMS identification for a file (FILENAME FILETYPE FILEMODE);
- if I le 0, the transmission takes place sequentially, according to the last input-output in the file; otherwise, I is the position in the file (first record has number 1);
- CLOSE must be used to close a file previously used for input (output), when now used for output (input); CLOSE may be used to ensure proper closing of a file.
- for further information about CMS disk I/O, see the ASSEMBLER macros FSOPEN, FSCLOSE, FSREAD, FSWRITE.

A3.4.6. Example of disk I/O. (See another example at 1.)

```
begin
     [] char FILE = "EX DATA A1";
     [1:100] char BUFFER; int POS;
     GETI(POS); GETS(BUFFER);
     # we update the POS-th record of the FILE, with BUFFER #
      DISKW(FILE,POS,BUFFER,1,100);
     # now, we make a listing of the FILE #
      CLOSE(FILE);
      proc EOF = (: begin CLOSE(FILE); STOP end );
     do (
          DISKR(FILE,0,BUFFER,1,100,EOF);
          # reading is sequential, because of the 0 #
          PUTS(BUFFER)
        )
end
```

A3.5. Non standard routines.

a. Reading a card

```
proc TAKE =
   (( ref [] char A, int I, J, proc EOF ):
    # This routine is equivalent to routine READ, but it tests
      for the CMS virtual end-of-file on the virtual reader; a
      logical end-of-file ('*EOF' characters) is also detected;
      when an end-of-file occurs, procedure EOF is called;
      limitations: I ge lwb A, I+J-1 le upb A, 0 lt J le 80.
      Runtime error code is that of FILL (12) or READ (30).
    #
  )
```

b. Ending a program

```
proc STOP = (: goto EXIT )

  # this procedure can be used to stop the execution of a program,
    or as parameter of the procedures TAPER, TAKE, DISKR, ... #
```

c. Standard end-of-file

```
proc EOF =
  (:
     begin PUTS("END OF FILE"); #printer#
           FORMAT(4,"S20"); PUT("END OF FILE"); #terminal #
           STOP
     end
  )
  # this procedure can be used in the same context as STOP #
```

d. Reading one data per card.

```
   proc GETI = (( ref int I):                                          #
        begin [1:80] char BUFFER ;

                  proc EOF = ( :  begin
                      FILL(BUFFER,1,80,"END OF FILE IN ""GETI""" );
                      PUTS(BUFFER); DISPLY(BUFFER,1,80); STOP
                                 end
                                ) ;
                  TAKE(BUFFER,1,80,EOF) ;
                  INI(BUFFER,1,80,I)
        end
              )

    proc GETLI = (( ref long int LI):
        begin [1:80] char BUFFER ;

                  proc EOF = ( :  begin
                        FILL(BUFFER,1,80,"END OF FILE IN ""GETLI""") ;
                        PUTS(BUFFER); DISPLY(BUFFER,1,80); STOP
                                 end
                               ) ;
                  TAKE(BUFFER,1,80,EOF) ;
                  INLI(BUFFER,1,80,LI)
        end
                  )

    proc GETR = (( ref real R):
        begin [1:80] char BUFFER ;

                  proc EOF = ( :  begin
                        FILL(BUFFER,1,80,"END OF FILE IN ""GETR""") ;
                        PUTS(BUFFER); DISPLY(BUFFER,1,80); STOP
                                 end
                               ) ;
                  TAKE(BUFFER,1,80,EOF) ;
                  INR(BUFFER,1,80,R)
        end
              )
```

```
   proc GETLR = (( ref long real LR ) :
        begin [1:80] char BUFFER ;

                    proc EOF = ( :  begin
                          FILL(BUFFER,1,80,"END OF FILE IN ""GETLR""") ;
                          PUTS(BUFFER); DISPLY(BUFFER,1,80); STOP
                                  end
                               ) ;
                    TAKE(BUFFER,1,80,EOF) ;
                    INLR(BUFFER,1,80,LR)
        end
                  )


   proc GETB = (( ref bool B):
        begin [1:80] char BUFFER ;

                    proc EOF = ( :  begin
                          FILL(BUFFER,1,80,"END OF FILE IN ""GETB""") ;
                          PUTS(BUFFER); DISPLY(BUFFER,1,80); STOP
                                  end
                               ) ;
                    TAKE(BUFFER,1,80,EOF) ;
                    INB(BUFFER,1,80,B)
        end
                 )


    proc GETS = (( ref [] char BUFFER):
        begin

                    proc EOF = ( :  begin [1:80] char BUFFER ;
                          FILL(BUFFER,1,80,"END OF FILE IN ""GETS""") ;
                          PUTS(BUFFER); DISPLY(BUFFER,1,80); STOP
                                  end
                               ) ;
                    TAKE(BUFFER, lwb BUFFER, upb BUFFER- lwb BUFFER+1,
                                 EOF)
        end
                  )
```

e. Writing one data per line.

```
     proc PUTI = (( int I):
        begin [1:20] char BUFFER ; OUTI(BUFFER,1,20,I) ;
                    WRITE(1,BUFFER,1,20)
        end
                    )

     proc PUTLI = (( long int LI):
        begin [1:20] char BUFFER ; OUTLI(BUFFER,1,20,LI) ;
                    WRITE(1,BUFFER,1,20)
        end
                     )
```

```
proc PUTR = (( real R):
     begin [1:30] char BUFFER ; OUTR(BUFFER,1,30,6,R) ;
                  WRITE(1,BUFFER,1,30)
     end
           )


proc PUTLR = (( long real LR):
     begin [1:30] char BUFFER ; OUTLR(BUFFER,1,30,14,LR) ;
                  WRITE(1,BUFFER,1,30)
     end
           )


proc PUTB = (( bool B):
     begin [1:10] char BUFFER ;
                  if B   then BUFFER := "     'TRUE'"
                         else BUFFER := "    'FALSE'"
                         fi ;
                  WRITE(1,BUFFER,1,10)
     end
           )


proc PUTS = (( [] char S):
     begin[lwb S :  upb S] char BUFFER ; BUFFER := S ;
                  WRITE(1,BUFFER, lwb S, upb S- lwb S+1)
     end
           )
```

f. Utilities.

```
proc ( ref [] char ) TIME =(( ref [] char A) :
  # This routine puts the time of day in A
    with the format  HH.MM.SS , left aligned and
    right filled with blanks  #
  # error if  upb A -  lwb A +1  gt 256  #
  # error if  upb A -  lwb A +1  lt 8    #
  # error code is 99 # )



proc ( ref [] char ) DATE = ((  ref [] char A ) :
  # This routine puts the date in A
    with the format MM/DD/YY , left aligned and right-filled
    with blanks #
  # error if  upb A -  lwb A + 1  gt 256  #
  # error if  upb A -  lwb A + 1  lt 8 #
  # error code is 98 # )
```

```
proc BLANK =
  (( ref [] char BUFFER ) :
        # This procedure fills a buffer with blank characters. #
  for I  from  lwb BUFFER  to  upb BUFFER  do ( BUFFER[I] := " " )
  )


proc DUMP =
     (( [] char A ):
       # prints a DUMP using the CP COMMAND 'DUMP';
         Example:  DUMP("EFFC.100")  #
     )

proc CMS =
     (( [] char COM, ref int I ):
      # executes the CMS command COM; I contains the RETURN-CODE
        Example:  CMS("LIST * * D",RC)  #
    )

proc CP =
     (( [] char COM, ref int I ):
       # executes the CP command COM; I contains the RETURN-CODE
         Example:  CP("MSG OP .......",RC)  #
     )
```

A4. Implementation characteristics.

A4.1. Compiler diagnostics.

No "compile-time diagnostics" are given in this manual, because of the completeness and clarity of the error messages printed during a compilation.

The "runtime diagnostics" are to be found in A7.

A4.2. Available characters.

Any character available on the hardware of the IBM 370 machine is accepted by the compiler (256 available internal characters).
No further limitations exist on tapes and disks and limitations on the available characters on special devices (TEKTRONIX,...) are to be found at another place. SET facilities of CP/CMS can also be used to represent characters.

A4.3. Equivalence between characters and integers.

Equivalence between characters available on the various devices which can be attached to a VM/CMS virtual machine are to be found elsewhere (3270, cable-print, EAI, Tektronics,...).

A4.4. Limitations on denotations.

1. Integral denotations are limited to 32767, but 'negative' integral values must be not less than -32768.

2. Long integral denotations are limited to 2147483647, but 'negative' long-integral values must be not less than -2147483648.

3. Real denotations:

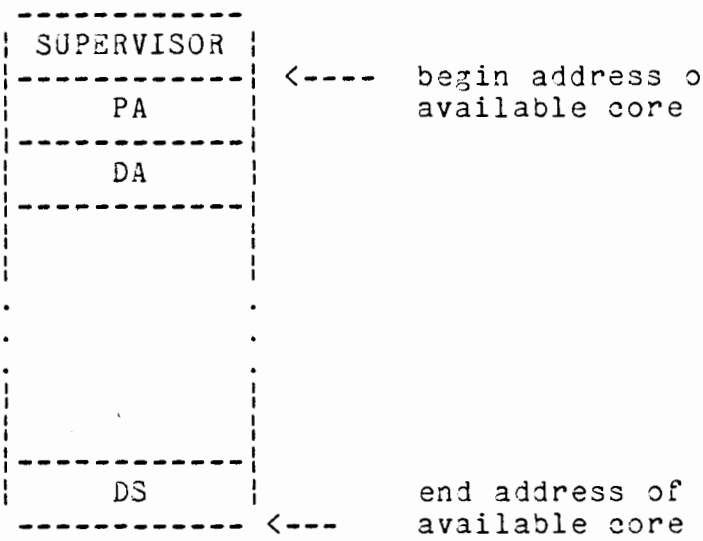
   -real: max 9 significant digits.
   -long real: max 18 significant digits.
   -exponent: max 2 digits.
   -if we normalize the number: .X1...Xn 10**s, X1 ne 0, then |s| le 75.

4. String denotations are limited to 127 characters. (This doesn't mean all row-of-character values are limited to 127: [1:20000]char A is executable, but FILL(A,1,20000,"...#200 characters#...") is syntactically incorrect).

A4.5. Runtime storage organization.

We will describe hereafter some general ideas on the "runtime storage organization" of this implementation of ALGOL68/19.

A4.5.1. General organization.



PA : program area

It contains the particular program, the separately compiled routines, if any, and (the part of) the standard prelude needed for the program; see the LOAD MAP for further information about the map of the PA.

DA : data area

It contains the 'static part' of the values of the identifiers used in the program; it is used as a 'stack' (crescent order of addresses in core).

DS : dynamic stack

It contains the 'dynamic part', if any, of the values of the identifiers used in the program; it is also used as a 'stack' (decrescent order of addresses in core).

Locations in the DA and DS are created at the elaboration of declarations and are freed when they are no longer accessible for the program (when leaving the range of that location).

R12 (General Purpose Register 12) is the base register used for PA; at any time of the elaboration of an ALGOL program, it contains the begin address of the current (external) procedure being executed (main program or separately compiled routine).

R13 (General Purpose Register 13) is the base register used for DA; at any time of the elaboration of an ALGOL program, it contains the begin address of the data area of the current procedure being executed (main program or procedure).

A4.5.2. Data areas.

(1) DA for a 'block' (begin -block or do -block)

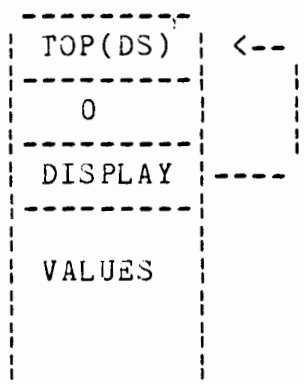
The DA for a block is aligned on a multiple of 4 bytes.

TOP(DS) is the current top of the dynamic stack (4 bytes); it changes dynamically when declaring multiple values.

TOP(DA) is static and gives the maximal address for the current DA, as long as we are in the reach of the block (4 bytes).

VALUES are the locations for the values of the identifiers declared in the reach of the block (see A4.5.3).

(2) DA for the 'program'



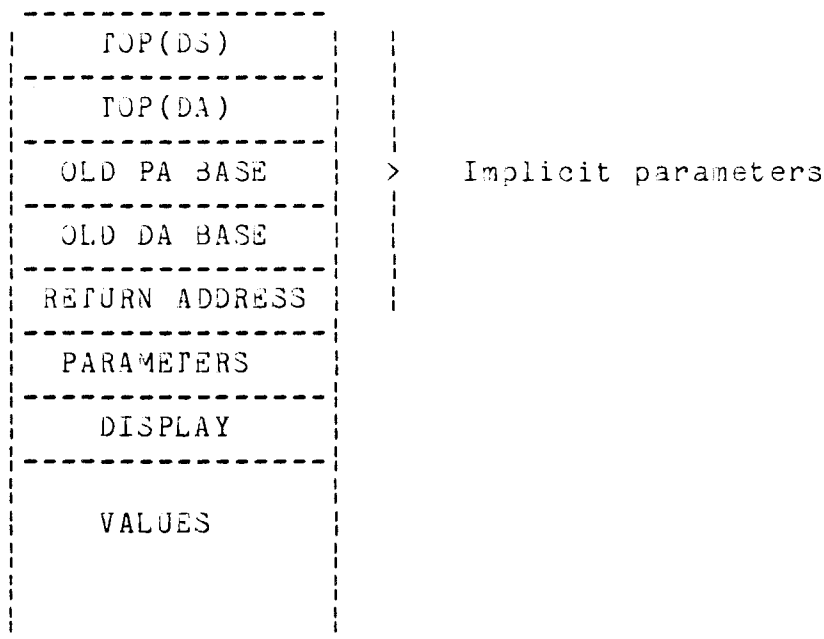
The DA for the program begins at the first available address after the PA (aligned on a multiple of 8 bytes).

TOP(DS) : last available address in core + 1

DISPLAY : address of this DA.

VALUES : see (1).

(3) DA for a 'procedure'



- Implicit parameters :

  TOP(DS) and TOP(DA) : see (1).

  OLD PA BASE : value of R12 of the 'calling' procedure.

  OLD DA BASE : value of R13 of the 'calling' procedure.

  RETURN ADDRESS : value of the return address to the 'calling' routine.

- Explicit parameters :

  Locations for the values of the parameters (see A4.5.3.).

- The DISPLAY is a list of addresses of the DA of the surrounding procedures (the address of the current procedure being the last in the DISPLAY). It is used to address locations of the values of the identifiers declared in procedures surrounding the current one.

  The length of the DISPLAY of a procedure is :
  ('level' of the procedure)*4 +4 bytes
  (Main program and separately compiled routine have level 0).

- VALUES : see (1).

A4.5.3. Representations of values.

1) Boolean values : last bit in one byte in DA.

2) Character values : 1 byte, in DA.

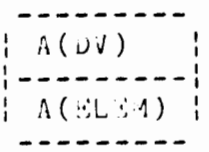
3) Integer values : 2 bytes, aligned on a 12, in DA.

4) Long integer values : 4 bytes, aligned on a 14, in DA.

5) Real values : 4 bytes, aligned on a M4, in DA.

6) Long real values : 8 bytes, aligned on a M8, in DA.

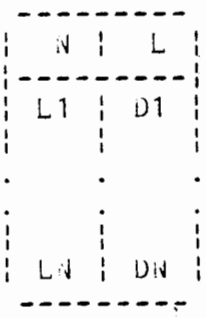
7) Multiple values : 8 bytes, aligned on a M4, in DA.



A(DV) is the address of the DOPE VECTOR.

A(ELEM) is the address of the elements, which stand in the DS.

DOPE VECTOR (standing in DA) :



N is the number of dimensions of the multiple value (2 bytes).

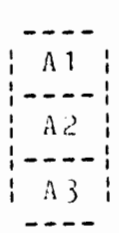
L is the total number of bytes required for the elements (2 bytes).

Li is the i-th lower bound (2 bytes).

Ui is the i-th upper bound. 1 le i le N

Di=Ui-Li+1 is the i-th stride (2 bytes).

8) Procedure value : 12 bytes, aligned on a M4, in DA.



A1 : address of the machine code for the procedure (4 bytes).

A2 : global display address (address of the DISPLAY of the surrounding procedure) (4 bytes).

A3 : global base register (base address of the PA of the separately compiled module (can be the main program) where

the declaration of the procedure occurs) (4 bytes).

9) Exceptions :

(1) a primitive denotation (integer, real, long integer, long real) stands in PA.
(2) the value of an external procedure : A1 in PA.
(3) a string denotation : elements in PA.
(4) a string denotation as parameter : (A(DV),A(ELEM),DOPE VECTOR)

A4.6. Example of an ASSEMBLER-compatible subroutine.

```
* WE ASSUME THIS ROUTINE IS CALLED FROM AN ALGOL PROGRAM;
* IT'S EQUIVALENT TO:
*
*     proc TEST = ( (int I, ref int J): J:=I )
*
*
ALTEST   CSECT
REG1     EQU   1
REG8     EQU   8
BASEREG  EQU   12
PARAMREG EQU   13
RETURNRG EQU   14
BRANCHRG EQU   15
*
         USING *,BASEREG
         USING PARAM,PARAMREG
*
* SAVE OLD REGISTERS AND SET NEW BASE REGISTERS:
*
         STM   BASEREG,RETURNRG,0(REG8)
         LR    PARAMREG,REG8
         LR    BASEREG,BRANCHRG
*
* BODY OF THE PROCEDURE;
* THE WHOLE SET OF REGISTERS CAN BE USED,
* EXCEPT BASEREG AND PARAMREG:
*
         L     REG1,J
         MVC   0(2,REG1),I
*
* RETURN TO THE CALLING PROGRAM:
*
         LM    BASEREG,RETURNRG,PARAM
         BR    RETURNRG
*
* SAVE AREA AND PARAMETER ZONE:
*
PARAM    DSECT
         DS    5A  SAVEAREA
I        DS    H
J        DS    A
         END
```

A5. Relations between the compiler and the VM/CMS-System.

A5.1. "Limitations" of the compiler.

ALGOL68/19 programs are punched in columns 1 to 80 of a CMS file with FILETYPE 'ALGOLCMS'. The compiler scans only columns 1 to 72; columns 73 to 80 may be used for numbering purposes.

Two compilers exist: FALGOL which is a very fast "compile and go" (executes immediately the source program) and doesn't accept separately compiled procedures nor common identifiers and ALGOLCMS which produces modules to be assembled by the CMS loader program.

Their principal limitations are:

- generated code for one module: at most 64K bytes (not including constants, variables and multiple values); see also 21.4 and 21.5;
- at most 64 blocks in a module;
- at most 256 identifiers in a block.

The compilers reside completely in core, which is one of the reasons for their fastness (no 'workfiles' on disks for tables,...); they require virtual machines of at least 182K bytes.

When the compilers are invoked, the input file must stand on a disk accessed in A, B/A, C/A...

A5.2. "Options".

- Option LIST causes the compiler to produce a listing (CMS file with FILETYPE 'LISTING';
- Option NOLIST suppresses the LIST option (no listing is produced, even if syntactical errors have been detected);
- Option PRINT causes the compiler to print the listing, if any, directly on the virtual printer;
- Option NOPRINT suppresses the PRINT option (the listing, if any, is a 'LISTING' file);
- Option XREF causes the compiler to produce a cross-reference and symbol table;
- Option NOXREF suppresses the XREF option;
- Option DECK causes the compiler to produce an object module (CMS file with FILETYPE 'TEXT') and COMMON files, if needed (see 21.5.);
- Option NODECK suppresses the DECK option (NODECK is the option taken if syntactical errors arises);
- Option TRACE causes the compiler to prepare a future trace at runtime (generation of a CMS file with FILETYPE 'ALGTRACE'; see Appendix 9;

- Option NOTRACE suppresses the TRACE option.

The "standard options" taken by the compiler are:

LIST, NOPRINT, NOXREF, DECK, NOTRACE.

A5.3. Calling the compilers.

The syntax of a call of the compilers is:

{ ALGOLCMS | FALGOL } <filename> [<option> {<option>}* [)]]

where <filename> is a CMS filename and <option> is one of the allowed options (see A5.2.).

A5.4. Relocatable libraries.

An 'ALG68LIB TXTLIB S2' library exists, which contains the standard prelude of ALGOL68/19. The CMS command 'GLOBAL TXTLIB ALG68LIB' must be used to load or include object modules produced by the ALGOLCMS compiler (this command is irrelevant for FALGOL). To use the facility of "AUTOLINK" offered by the CMS loader, all separately compiled procedures must be edited in CMS files with filename 'ALXXXX..' and filetype 'ALGOLCMS', if the name of the procedure is 'XXXX..'. Otherwise, the programmer can obviously choose the filename.

Example:

proc ROUT1 = (: ....) must be edited as a file 'ALROUT1 ALGOLCMS', if you want the autolink of ROUT1; if not, you must type the INCLUDE command.

The programmer can use the facilities of CMS to catalogue object modules in private 'TXTLIB's. Like any other separately compiled routine, catalogued routines are available in ALGOL68/19 programs with the use of "pragmats".

A5.5. Linkage editing.

The order of compilation or loading of separately compiled programs is meaningfull. First 'LOAD' is considered to be the main program. If you want to change the order in the loading, see the LOAD command.

A5.6. Core image modules.

Use the CMS command GENMOD to catalogue a previously loaded program. Note that in ALGOL, you must specify the entry point when using GENMOD, if the name of the module is choosen to be different from the name of the main program.

Example:

1) LOAD TEST
   GENMOD TEST(FROM ALTEST
   or
   LOAD TEST
   GENMOD ALTEST

2) LOAD TEST
   GENMOD TEST(FROM ALMAINPG
   or
   LOAD TEST
   GENMOD ALMAINPG

if TEST ALGOLCMS contains:

TEST: begin skip end begin skip end

A6. "Common" identifiers.

Separately compiled procedures may use mode-identifiers defined in the outer begin-block of some main-program. This is achieved in two steps:

- defining "labeled commons" or "common blocks" to be saved at the end of a main program;

- using pseudo common declarations at the beginning of a separately compiled procedure.

a. Saving a labeled common.

After the end-symbol of a main program, the programmer can add a <saved common>:

<saved common>::= SAVE <common block> {,<common block>}*

<common block>::= {<common identifier> ALL |
{<common identifier>}0 (<id>{,<id>}*) }

<common identifier>::= <id> (see Appendix 2)

<Common identifier> will be the name of a <common block>. When not present, the FILENAME of the current compiled program is taken as <common identifier>. In a <saved common>, at most one <common identifier> may be taken to be optional. <Common identifier> may not be 'ALL'.

<Id> must be a mode-identifier (no labels are allowed (1)) defined in the reach of the outer begin-block of the current main program. If 'ALL' is used, the whole set of identifiers declared in the reach of the outer begin-block is considered to be saved in the current <common block>. The order of the identifiers has no special meaning.

(1) See also 21.6, to have a label in common.

Note that option DECK must be active to ensure proper saving of commons. The compiler uses and saves CMS files with FILETYPE 'COMMON' to compile common blocks.

b. Using a labeled common.

Before a separately compiled procedure, pseudo declarations may be added, using the special "comment" surrounded by co-symbols (see 7.). These declarations are considered to be added to the library prelude.

The syntax of this <common comment> follows:

<common comment>::= co <common item> {,<common item>}* co

<common item>::= <common identifier>
{ ({<id1>=}0 <id2> { , {<id1>=}0 <id2> }*) }0

<common identifier>::= <id1>::= <id2>::=<id> (see Appendix 2)

<Common identifier> must be the name of a <common block> defined in some main program. When the optional part of the <common item> is not present, the whole set of identifiers of this <common block> is used, i.e. all identifiers present in the <common block> can be used in this procedure.

Otherwise, <id2> must be an identifier contained in the current <common block> and <id1> is the new identifier associated with <id2>, i.e. <id1> can be used in this procedure instead of <id2>. If <id1> is not present, it's considered to be the same as <id2>.

The number and order of the identifiers have no special meaning. The mode of the identifiers is considered to be the same as the mode of the corresponding identifiers in the current <common block>.

c. Security when using common identifiers.

Absolute security is given to the programmer when using the common feature: when in a main program, a <common block> is saved and it doesn't 'correspond' (1) with a <common block> having the same <common identifier> defined in a previous compilation of this main program, all TEXT files of procedures using this common are erased; furthemore, in separately compiled routines using <common items>, modes of <common identifiers> are tested for compatibility.

To ensure this security, you must compile the main programs defining <common blocks> before compiling subroutines using them. Otherwise, the compiler will reject any attempt to use a not yet saved common. Furthemore, when re-compiling main programs (because of modifications to declarations and/or to <common blocks>, you would first erase COMMON files; otherwise the compiler will probably find syntactical errors.

The declarations of identifiers saved in a <common block> must have been elaborated at runtime before they can be used in a procedure using this common (see 27.1. Context conditions).

(1) To 'correspond' means that, for a given identifier, the mode and displacement in the Data Areas at runtime are the same in the two <common blocks> (see A4.5.).

d. Examples.

1) Main program:

```
# we assume this program is edited under
  the name of PROG1 ALGOLCMS A1 #

pr proc P1, proc(ref int) P2, proc P3, P4 pr

begin
     .....
     int A1 = 5;
     .....
     int A2; ... [A1:A2] real MAT; ...
     .....
     proc TT = (: ..... );
     .....
     bool B;
     .....
     P1; ..... P2(A2); ..... P3; ..... P4; .....
end

SAVE COM1 ALL , COM2(A1,A2) , COM3(B,TT), ALL
```

2) Subroutine 1:

```
co COM1 co

proc P1 = (:
   # this routine can use A1, A2, TT, B, MAT,... without
     declaring them #
        )
```

3) Subroutine 2:

```
co COM2(A1,A2) co

proc P2 = (( ref int A3):
  # this routine can use A1, A2 without declaring them;
    note that modifying A3 changes also the value of A2 #
         )
```

4) Subroutine 3:

```
co COM3(TT1=TT) co

proc P3 = (:  .... TT1 ....  )
  # this has the same effect as calling TT #
```

5) Subroutine 4:

```
#(1)# co PROG1 co      #or (2)# co PROG1(A3=A2,A1) co

proc P4 = ( :
    # this routine can use the whole set of identifiers
      of the main program (1), or only A1 and A3 (A3 means A2)
      (2) of the main program #
          )
```

A7. Runtime errors.

A7.1. Undetected errors.

Some execution time errors are not detected. They are :

- scope condition not satisfied in an assignation ;
- in a multiple value declaration for A, the condition

  -32K le Ui-Li+1 lt 32K

  is not satisfied for each (Ui, Li) of A ;
- In a multiple value declaration for A, the total number of bytes required for the elements of A is at least 32K ;
- undefined formula evaluation ;
- uninitialized values ;
- use of identifiers whose declaration has not yet been elaborated (see 27.1, Context conditions).

These errors can cause unpredictable situations, in particular "PROGRAM CHECKS", at the time the error occurs, or later.

A7.2. Detected errors.

Some other errors (see A7.5) in the standard and non-standard routines are detected at run time. They are handled by a module named "AL$ERROR", not accessible for the ALGOL programmer.

This module prompts at your terminal the message

```
RUNTIME ERROR XX ( message )
```

and writes on the virtual printer the message

```
RUNTIME ERROR XX ( message )
AT UNIT YY OF BLOCK ZZ
PROGRAM BASE REGISTER = X'TTTT'
DATA BASE REGISTER = X'DDDD'

TRACEBACK FOLLOWS :
```

XX is the code of the error (see A7.5) ;

YY is the unit number af the ALGOL-unit where the error occurs (see A7.3) ;

ZZ is the block number of the ALGOL-block where the error occurs (see A7.3) ;

TTTT is the begin address of the 'current' separately compiled routine (main or procedure) which has caused the error (code 0 or 1), or which has called the standard routine which causes the error (other codes).

Message is a short description of the error.

To avoid printing of these messages on the real printer, set the CP command 'SPOOL PRT TO *' in your 'PROFILE'.

The action after printing the error messages is: goto EXIT , but see also the recovery of runtime errors in A7.7.

A7.3. Blocks and units.

A "block" is a begin -block, a do -block, or a procedure. Blocks are numbered in the order their begin-symbols appear ; this is the same numbering as in the LISTING file. Blocks are numbered in the order they appear in the program. Block number of a main program or of a separately compiled routine is 1.

A "unit" is:

- the 'opening of a block or procedure' ;
- a statement, except if-statement and case-statement ; or
- a declaration (between ;) ; or
- an ' if part' ; or
- a ' case part' ; or
- a ' for - from - by - to - while -part' (even if not physically present), which is unit 1 of the do-loop; or
- a parameter of the GET and PUT procedures (see FORMATTED I/O) .

Units are numbered in the order they appear in each block. The 'opening of a block' has unit number 0.

A7.4. Runtime error localisation.

To locate precisely a runtime error in a program :

- consider the runtime error message (A7.2) ;
- look in the LOAD MAP for the module with begin address TTTT ; this is the module where the error has occured ;
- count the number of blocks and units to locate the unit where the eror has occured (use the LISTING file which contains the block numbers).

If a "program check" occurs, AL$ERROR will also be called. This module prompts at your terminal the message

PROGRAM CHECK (message)

and writes at the virtual printer the message

```
PROGRAM CHECK XX (MESSAGE) INSTRUCTION ADDRESS = X'AAAA'
AT UNIT YY OF BLOCK ZZ
PROGRAM BASE REGISTER = X'TTTT'
DATA BASE REGISTER = X'DDDD'
TRACEBACK FOLLOWS :
```

XX is the code of the "program check" :

01 : operation exception;
02 : privileged operation;
03 : execute;
04 : protection exception;
05 : addressing exception;
06 : specification exception;
07 : data exception;
08 : fixed point overflow exception;
09 : fixed point divide exception;
10 : decimal overflow exception;
11 : decimal divide exception;
12 : exponent overflow exception;
13 : exponent underflow exception;
14 : significance;
15 : floating point divide exception.

INSTRUCTION ADDRESS is the virtual machine address of the error.

MESSAGE, XX, YY, ZZ, TTTT, DDDD and TRACEBACK FOLLOWS have the same meaning as for the 'detected errors'.

To avoid printing of these messages on the real printer, set the CP command 'SPOOL PRT TO *' in your 'PROFILE'.

If an "interrupt" occurs (caused by the virtual "interrupt-key"), AL$ERROR will also be called; the message is:

INTERRUPT AT UNIT YY OF BLOCK ZZ .....

The virtual interrupt key is the CP command EXTERNAL; this can be usefull when a program "loops" infinitely or to cause ALGOL trace to be entered.

If an error 4 has occured during code generation, the programmer must know that units are numbered modulo 255.

A7.5. Codes of execution-time errors.

00 There is not enough main storage to execute the program (too large matrices, too large data area, partition F1 or F2 too large, ...)

01 Subscript is not between the declared limits, or the descriptors on both sides of an assignation of a multiple value are not identical.

ERRORS IN STANDARD ROUTINES (for more detail, see appendix 3)

10 : COMP	11 : TRC	12 : FILL	13 : INB	14 : OUTB
15 : INI	16 : INLI	17 : OUTI	18 : OUTLI	19 : INR
20 : INLR	21 : OUTR	22 : OUTFR	23 : OUTLR	24 : OUTFLR

50 : EXP	51 : LN	52 : SQRT	53 : SIN	54 : COS
55 : TAN	56 : ARSIN	57 : ARCOS	58 : ARTAN	59 : LEXP
60 : LLN	61 : LSQRT	62 : LSIN	63 : LCOS	64 : LTAN
65 : LARSIN	66 : LARCOS	67 : LARTAN		

30 : LINE	30 : PAGE	31 : READ	32 : PUNCH	33 : ACCEPT
34 : DISPLY	35 : TAPER	35 : TAPEW	36 : CLOSE	37 : DISKR
37 : DISKW	38 : terminal EOF		70-77 : formatted I/O	

ERRORS IN NON-STANDARD ROUTINES.

96 : RESET	97 : ON	98 : DATE	99 : TIME

A7.6 Deleted

A7.7. Runtime errors recovery.

The whole set of execution errors (input-output errors, conversion errors, end of file in formatted input-output, errors in mathematical functions,...) can be bypassed at runtime.

For that purpose, the following declarations are added to the standard prelude (available without pragmats).

```
proc ON =
      (( int CONDITION TO BE BYPASSED #,
         proc PROCEDURE TO BE EXECUTED #
      ) :
       begin
           # COND is an integer whose value is the code number
             of the condition to be recovered ; it's value can
             only be one of the detected 'runtime errors',
             except runtime errors 0 and 1.
             PROC is the procedure which will be elaborated
             when this error occurs.
           #
       end
      ) ;
```

```
proc RESET =
      (( int CONDITION TO BE RESET# )
        # COND is the code number of an error;
          the system resets the standard action for
          this error
        #
      )
```

Example :

```
TEST :  begin
          do
          (
             real X, Y; Y:=0.0;

             int EOF = 77 ;
             proc EOF1 = ( :  goto LABEL1 ) ;
             ON(EOF,EOF1) ;

             int NEG = 52 ;
             proc ERROR =
             ( :
                    Y:= -999.0
             ) ;
             ON(NEG,ERROR) ;

             FORMAT(1,"*(F8.4)") ;
             do ( GET(X); Y:=Y+X ) ;

              # when the end of file occurs, the loop ends
                and we go to LABEL1 #

   LABEL1 :  Y:=SQRT(Y) ;

              # if Y is negative, a special value will be
                computed for Y and the program will continue
                normally #

             RESET(NEG); # no programmer action for
                errors in SQRT #
             PUTR(Y)
          )
        end
```

Notes :

- The programmer must be very carefull with the choice of the procedure he passes as parameter to ON and RESET (see Scope Conditions (27.5)). The further elaboration of the program may be 'undefined' as in the next example :

Example :

```
TEST :  begin
            begin
                proc A=(:  goto L) ;
                ON(77,A) ;
              L :  skip
            end ;
            [1:100] int I,J ; FORMAT(1,"*(I4)") ;
            GET(I,J)  #if in this statement an end of file oc-
            curs, the program violates the scope conditions and
            the further elaboration is undefined #
        end
```

- If the programmer doesn't use these recovering facilities, the standard action will be taken.

- The total number of the bypassed errors cannot exceed 16 at any time of the elaboration of an ALGOL68/19 program.

A6. Symbols and their representations.

letter	A to Z (capitals)	over	over
digit	0 to 9	modulo	mod
point	.	lower bound	lwb
divided by	/		entier
up	**	upper bound	upb
plus	+	not	not
minus	-	absol. value of	abs
times	*	representation of	repr
becomes	.= or :=	lengthen	leng
cast of	.. or :	shorten	short
open	(	odd	odd
close	)	sign	sign
comma	,	round	round
sub	(/ or < or [	integral	int
bus	/) or > or]	real	real
up to	:	boolean	bool
label	.. or :	character	char
go on	., or ;	long	long
quote	' or $	reference to	ref
equals	eq in formulas	procedure	proc
	= otherwise	begin	begin
comment	pr before a program	end	end
	co before a program		
	#,% at any place	if	if
true	true	then	then
false	false	else	else
or	or	fi	fi
and	and	goto	goto
not equal	ne	skip	skip
is less than	lt	for	for
less or eq.	le	from	from
gr. or eq.	ge	by	by
greater than	gt	to	to
times ten to		do	do
the power	E	while	while
case	case	in	in
out	out	esac	esac

Keywords can be punched in two ways: with or without "apostrophes" ('): begin for example, can be represented as BEGIN or 'BEGIN'; but don't forget that in the CMS version of ALGOL68/19, keywords cannot be used as identifiers.

A9. Calling FORTRAN subroutines.

A9.1. Generalities.

The possibility exists to call FORTRAN external subroutines within ALGOL programs. Therefore, the programmer must add 'FORT' before each virtual declaration of a FORTRAN subroutine in the 'pragmat' (21.4) before his program.

Example:

```
pr FORT proc A1, A2, proc(int) A3  pr
pr FORT proc(int,ref int,ref[,]real) A4  pr
          begin .... end
```

means that:

- two external FORTRAN subroutines without parameters are available;
- an external ALGOL procedure exists;
- an external FORTRAN subroutine with three parameters is available.

The compiler provides a test to see if the virtual parameters are of one of the following modes:

- int, long int, real, long real, char, bool   (1)
- ref int, ref long int, ref real, ref long real, ref char, ref bool
- []char, ref[]char
- { ref }0 [{,}*] 'one of the modes (1)'

When encountering such pragmats, the compiler generates an ALGOL-FORTRAN interface: when option DECK is active, it generates a TEXT file ALxxxx, if the name of the external FORTRAN subroutine is 'xxx'.

A9.2. Correspondence between ALGOL and FORTRAN parameters.

	corresponds to
{ ref }0 int	INTEGER*2
{ ref }0 long int	INTEGER*4
{ ref }0 real	REAL*4
{ ref }0 long real	REAL*8
{ ref }0 char	LOGICAL*1
{ ref }0 bool	LOGICAL*1
{ ref }0 []char	LOGICAL*1 variable (n) where n is the number of elements of the ALGOL value.
{ ref } [{,}*] int	INTEGER*2 variable (N1,...Np) where p is the number of dimensions of the value and the Ni are the number

of elements in each dimension.

Similar correspondence for long int, real, long real, bool, char.

A9.3. Important notes.

1) The philosophy of the use of ALGOL parameters is preserved by the interface: nevertheless, an ALGOL parameter passed without ref can have its value modified by the FORTRAN subprogram; if an ALGOL parameter is passed with a ref, the FORTRAN subroutine can modify the referred value.

Examples:

```
 pr FORT proc(int) EXT1  pr          SUBROUTINE EXT1 (I)
begin                                INTEGER*2 I
  int B; B:=0;                       I=I+1
  EXT1(1);                           RETURN
  EXT1(B); # B eq 0 #                END
end
```

The value of B and that of 1 are not modified by EXT1.

```
 pr FORT proc(int,ref int) EXT2 pr
begin                                SUBROUTINE EXT11 (I,J)
   int I;                            INTEGER*2 I,J
   EXT2(1,I); # I eq 2 #             J=I+1
   EXT2(I,I); # I eq 3 #             RETURN
end                                  END
```

2) Note that if the types and numbers of FORTRAN parameters doesn't map the modes and number of ALGOL parameters, the effect would be undefined.

3) The i-th dimension of a FORTRAN parameter corresponds to the number of elements of the (n-i+1)-th <bound pair> of the ALGOL parameter.

Example: [-3:3,1:4] real A corresponds to REAL A(4,7)

4) The correspondence between ALGOL char and bool values and FORTRAN LOGICAL*n variables is imperative. If another types are chosen, the effect is undefined (this is often a question of alignment).

Example:

```
pr FORT proc([]char)EXT pr
begin EXT("abcd") end
```

```
      SUBROUTINE EXT(A)
      LOGICAL*1 A(4)  THIS IS THE NORMAL DECLARATION
C     LOGICAL*4 A     THIS DECLARATION IS PERMITTED
C     REAL A          IF THIS DECLARATION IS USED, AN ERROR
C                     WILL PROBABLY OCCUR.
      WRITE(6,*)A
      RETURN
      END
```

5) Don't forget ALGOL multiple values are stored 'line per line', whereas FORTRAN arrays are stored 'column per column'.

Example:

```
pr FORT proc(ref[,]long int) EXT3 pr
begin [1:2,1:3] long int A;
      EXT3(A);
      FORMAT(4,"stream"); PUT(A)
end
  # the displayed values are:
    1 2 3 10 5 6
  #
```

```
      SUBROUTINE EXT3(A)
      INTEGER A
      DIMENSION A(3,2)
      READ(5,*) A
C     the data entered
C     from the terminal are:
C      1 2 3 4 5 6
      A(1,2)=10
      RETURN
      END
```

6) Bounds of multiple values as parameters must be of the mode long int. This is a FORTRAN restriction: it doesn't accept at run time bounds of arrays to be INTEGER*2 variables.
See A9.4 for examples.

7) Recuperation of execution errors.

When an execution error due to a FORTRAN subprogram occurs, two cases arises:

- if it's a program check, ALGOL recuperates the interrupt and the following message will be printed:

```
IN A FORTRAN SUBPROGRAM...
  PROGRAM CHECK nn ( message)
  INSTRUCTION ADDRESS = XXXXX
  CALLED AT UNIT XX OF BLOCK YY
      ........     see A7
```

- if it's another runtime error (for example computing the SQRT of a negative number or invalid FILEDEF,...), the message will be printed by FORTRAN and the program will terminate with ABEND.

8) Don't forget the CMS command GLOBAL TXTLIB FORTLIB ALG68LIB when loading an ALGOL-FORTRAN mixed program.

A9.4. Simple example.

```
pr FORT proc(ref[,]real,long int,long int,ref real)SUM pr

    begin
        int N, M; GETI(N); GETI(M);
        [1:N,1:M]real MAT1;
        FORMAT(1,"*(F10.0)"); GET(MAT1);
        real RES1, RES2;
        SUM(MAT1,leng N,leng M,RES1);
        RES2:=0;
        for I to N do ( for J to M do ( RES2:=RES2+MAT1[I,J] ) );
        if RES1 ne RES2 then
        then # c'est a douter de tout! # stop
        fi
        [1:2,1:3] real MAT2;
        for I to 2 do
        ( for J to 3 do ( MAT2[I,J]:=I+J )
        );
        SUM(MAT2,long 2,long 3,RES); PUTR(RES)
    end


    SUBROUTINE SUM(MAT,N,M,RES)
    REAL*4 MAT(M,N)
    INTEGER N,M
    DO 1 I=1,M
    DO 1 J=1,N
1   RES=RES+MAT(I,J)
    RETURN
    END
```

A10. Formatted input-output.

A10.1. Introduction.

The standard prelude contains three procedures dealing with formatted input-output.

They are :

- proc(int ,[] char) FORMAT

- proc([] union (int, real, long int, long real, char, [] char, ref [] int, ref [] long int, ref [] real, ref [] long real, ref [,] int, ref [,] long int, ref [,] real, ref [,] long real)) PUT

- proc([] union (ref int, ref long int, ref real, ref long real, ref char, ref [] char, ref [] int, ref [] long int, ref [] real, ref [] long real, ref [,] int, ref [,] long int, ref [,] real, ref [,] long real)) GET

For the programmer who is not familiar with the ALGOL68 notations, a procedure with a parameter of mode [] union (mode1 ,..., mode N) is approximatively a procedure accepting a variable number of parameters, each of which being of one of the modes mode I. The only available coercion on each parameter is the dereferencing.

FORMAT is a procedure by which a <format> is associated with an 'input-output device'.

PUT and GET are procedures by which data (ALGOL values) are read or written under the control of a <format>.

A10.2. Input-output devices.

The first parameter of the FORMAT procedure is an integer value giving the 'input-output device', associated with the <format>.

If its value is 1, the device is the virtual reader.

If its value is 2, the device is the terminal, used as input.

If its value is 3, the device is the virtual printer.

If its value is 4, the device is the terminal, used as output.

In the rest of this chapter, we shall denote by 'IO' the input-output device associated with a <format>.

There exists two 'types' of devices:

- input devices (1 and 2)
- output devices (3 and 4).

A10.3. Syntax of <formats>.

The second parameter of the FORMAT procedure is a 'row of character' value giving the <format>.

```
<format>::=   { STREAM | { <record length>,}0
          {<item> {,<item>}* | {<item>,}* <stream group>} }
<stream group>::=     * ( <item> {, <item>}* )
<item>::=      { <simple item> | <group> }
<group>::=     <number> ( <item> {, <item> }* )
<simple item>::=  { <alphanumeric code> | <control code> }
<alphanumeric code>::=   { <integer code> | <real code> | <string code> }
<integer code>::=    I <number>
<real code>::=       {E | F } <number>.<number0>
<string code>::=     S <number>
<control code>::= {<line code> | <page code> | <skip code> | <column
                    code> | <no skip line code> }
<line code>::=       L
<page code>::=       P
<skip code>::=        X <number>
<column code>::=    C <number>
<no skip line code>::= R
<number>::=        {d}+
<number0>::=       {d}+
<record length>::=   <number>
<d>::=              {0|1|2|3|4|5|6|7|8|9}
```

Blanks have no special meaning in a <format>.

In order to improve readability or construction of <formats>, blanks may be used freely.

In the rest of this chapter, when we shall speak about <formats>, we shall suppose <formats> have been 'developed' : to 'develop' a <format> we replace all <groups> in it by a number of times (corresponding to <number>) the list if <items> contained in that <group> and we restart the process.

Then, the notation 'next item in a format' will become obvious.

Example : I4,3(I5,S4) is developed into I4,I5,S4,I5,S4,I5,S4

A10.4. General semantics of a <format>.

When a call of the FORMAT procedure occurs, a <format> is made 'available' and is associated with its input-output device.

This <format> will remain 'available' until another call of the FORMAT procedure for an input-output device

of the same type is elaborated.

In other words, at any time of the execution of an ALGOL program, at most two 'available' <formats> exist (one for input devices (1 and 2) and one for output devices (3 and 4)).

Furthermore, with any <format> is associated a 'buffer' of a certain 'length'. The buffer length is the number of characters which will be considered on each card or on each printer or terminal line. They are taken in the leftmost positions of the card or of the line.

If no <record length> is specified in the <format>, the value of 'length' is 80 when IO=1, 130 when IO=2, 132 when IO=3 and 130 when IO=4 (80, 130 and 132 are the maximal allowed values).

A10.5. General semantics of the GET and PUT procedures.

When a <format> is 'available',

- if IO = 3 or 4, sequences of calls of the PUT procedure can be elaborated.
- if IO = 1 or 2, sequences of calls of the GET procedure can be elaborated.

Given this sequence of calls using the same <format>, consider the virtual sequence composed of all parameter lists together. In this last sequence (refered to as the 'data list'), the notion 'next item in the data list' becomes obvious.

The case of multiple values is explained later on (see A10.10).

To elaborate the sequence of calls of GET (resp. PUT), we consider together

1) the sequence of <items> of the available <format>
2) the sequence of <items> of the 'data list'

Then, we 'elaborate' (cfr later on) the first <command codes> of the <format>, if any, until we reach an <alphanumeric code>.

Then, for each data,

- we 'elaborate' (cfr later on) the 'transmission' of this data under the control of the <alphanumeric code>.
- we 'elaborate' the following <command codes> until we reach an <alphanumeric code> or the end of the <format>.

The transmission takes place between the program and the buffer. It ends when the end of the data list of a particular GET or PUT instruction is encountered. It may be resumed if a new GET or PUT instruction is elaborated with the same <format>.

The 'buffer' will be filled according to the 'elaboration' of the sequence of the <data items> of the data list. It will be filled from left to right ; each <alphanumeric> transmission occupies the next w (see A10.6) characters in the buffer, the 'first item' in the

sequence beginning at the first position.

An error condition is raised (see A10.11) if there is no more <alphanumeric code> for a given <data item>, or if there is not enough space in the buffer for the next data item.

The transmission between the buffer and the input-output device is governed by the following rules :

- on input, the buffer is filled with the next record at the elaboration of the first call of GET. It may be filled again when the end of a <stream group> is encountered (see A10.7 f), or if an L code is used (see A10.7 a).

- on output, the buffer is printed or displayed if an L or P code is encountered, or if the end of a <stream group> is reached (see A10.7 f), or if the format ceases to be available, or at the end of the program.

A10.6. The transmission of data under the control of an <alphanumeric code>.

a) Integer code.

The I format code is used to transmit data of mode int or long int .

Its form is "Iw", where w is a positive number giving the total number of characters involved in the transmission.

The conversion rules and restrictions on the value of w and on the format of the transmitted data are to be found in the procedures OUTI, OUTLI, INI, INLI.

b) Real code.

The E and F format codes are used to transmit data of mode real or long real .

Their form is "Fw.d" or "Ew.d", where w is a positive number giving the total number of characters involved in the transmission and d is a (possibly 0) positive number giving the number of digits after the decimal point.

The conversion rules and the restrictions on the values of w and d and on the format of the transmitted data are to be found in the procedures OUTR, OUTFR, OUTLR, OUTFLR, INR, INLR.

On input, only w is significant (see INR and INLR).

c) String code.

The S format code is used to transmit data of mode char or [] char .

Its form is "Sw", where w is a positive number giving the total number of characters involved in the transmission.

If the transmitted data is a character, then the S format code specifies a field of w characters : on input, the first character of the field is read and the next (w-1) are skipped ; on output the character is written and the next (w-1) are filled with blanks.

If the transmitted data is a row of character of length l, then the S format code specifies a field of w characters : on input, the first l characters of the field are read and the next (w-l) are skipped ; on output, l characters are written and the next (w-l) are filled with blanks.

If l>w then a runtime error number 73 occurs.

A10.7. Control codes.

a. Line code.

The L format code is used as control code.
If IO = 1, the transmission of data begins at the next card.
If IO = 2, 3 or 4, the transmission of data begins at the next line.

b. Page code.

The P format code is used only when IO = 3.
Transmission of data begins at the next page.

c. Remain code.

The R format code is used only when IO = 3.
It allows the programmer to make "overwriting".
The transmission of data begins at the same line of the printer.

d. Skip code.

The form of the skip code is "Xw", where w is a positive number.
The X format code specifies a field of w characters to be skipped on input or filled with blanks on output.

e. Column code.

The form of the column code is "Cw", where w is a positive number
The C format code specifies the position in the record where the transfer of the next data is to begin.

C may be used to "backspace" (see examples).

f. Stream group.

The stream group has two effects :

(1) if format control reaches the end of a <stream group>, then control reverts to the beginning of that <stream group>, permitting infinite replication of formats.

(2) data transmitted under the control of a <stream group> are considered to be "stream oriented". This means that data

may extent beyond the capacity of the buffer. On output, when it becomes full, it is transmitted and used again for the next data, if any. On input, when it has been read into the last position, it is filled again with the next card, if needed.

Notes :

1. Even in a <stream group> transmission, the programmer can bypass the point (2) (with control codes) and use only the facility of infinite replication.

2. No data item may be truncated by the end of the buffer (see A10.10.4).

A10.8. Transmission of data without format.

The possibility exists to transfer data 'without' format, using the GET and PUT procedures. Data are then considered to be 'stream oriented' and the real format has no special meaning.

Example:

```
FORMAT(1,"stream"); GET(I,J);
FORMAT(4,"stream"); PUT(A,B);
```

This involves the transmission of data with a standard format on output and with no format on input. On input, a comma (,) is used to separate data and blanks may be used freely. No data may be truncated between two lines.

This feature is similar to the GET LIST and PUT LIST of PL/I and to the unformatted input-output of FORTRAN.

A10.9. Simple example.

```
SIMPLE#EXAMPLE OF FORMATTED INPUT - OUTPUT# :
   begin
         #
          we want to read a matrix, line per line;
          the first card contains the dimensions of the matrix
          and the next ones the elements ( 10 per card )
         #
         [] char FORM#AT#1 = "2(I4),L,*(F6.2,X2)" ;
         int N , M ;
         FORMAT(1,FORM#AT#1) ; # a new <format> is 'available'
                                  for the reader #
         GET(N,M) ;  # we read the bounds with the '2(I4)' <item> #
         [1:N,1:M] real A ; # we reserve dynamically the place #
         GET(A) ; # reading the matrix with the '*(F6.2,X2)' <item> #

         #-------------------------------------------------------------#

         # now, we read a list of variables #

         int A1 , A2 ;  real A3 , A4 ; [1:60] char BUFFER ;
         [1:3] long int A5 ;
         FORMAT(1,"I4,C50,I10,L,S60,L,3(X5,I10),L,F5.0,C1,F5.0") ;
         GET(A1,A2,BUFFER,A5,A3,A4) ; #A3 and A4 possess the same value#
         # or  GET(A1,A2,BUFFER,A5) ; ..... ; GET(A3,A4) ;  #

         #-------------------------------------------------------------#
         #
           another matrix is read , column per column
           with a format given by the programmer at runtime
         #

         GETI(N) ; GETI(M) ;
         [1:80] char FORM#AT#2 ; [1:N,1:M] real B ;
         GETS(FORM2) ; # the <format> is punched freely on a card
                         and we suppose we read :
                            72,*(F6.2,X2)
                       #
         FORMAT(1,FORM#AT#2) .,

         for I  to M  do (  for J  to N  do ( GET(B[I,J]) ) ) ;

         #-------------------------------------------------------#

         [1:100] int C ;  now we use 2 <formats>, one for the
                                reader and one for the printer #
         FORMAT(1,"STREAM");
         FORMAT(3,"120,*(I4,X4,I4)") ;
         for I  to 100  do ( GET(C[I]) ; PUT(I,C[I]) )

   end # of this simple program #

# note that three ways of defining a <format> are given #
```

A10.10. Important notes.

1) Facilities for input-output of multiple values have also been provided.

When two-dimensional arrays are used, the transmission of data is performed line per line. If transmission column per column is needed, the programmer must compute himself the indexes.

example :

```
FORMAT (3,"*(I4)") ;
[1:N] int MAT1, MAT2 ;
PUT(MAT1,MAT2) ;  # has the same effect as #
for I  to N  do ( PUT(MAT1[I]) ) ;
for I  to N  do ( PUT(MAT2[I]) ) ;
[1:N, 1:M]  int  MAT3 ;
PUT(MAT3) ; # has the same effect as #
for I  to N  do (  for J  to M  do ( PUT(MAT3[I,J]))) ;
                # but has not the same effect as #
for J  to M  do (  for I  to N  do ( PUT(MAT3[I,J]))) ;
```

2) The possibility of reading, constructing and transforming <formats> at runtime is provided : see the examples in A.10.9.

3) <Formats> have nothing in common with the block structure mecanism of ALGOL : they are 'global' in the sense that, when a <format> is 'available', opening and closing of blocks or calling of procedures have no effect on the fact that a <format> remains or not 'available'.

The following program is perfectly executable :

```
TEST :  begin    int I , J ;
                 proc A = ( : PUT(I) ) ;
                 FORMAT(3,"stream");
                 begin  int J ; PUT(J) ;
                        A ;
                        FORMAT(1,"10(I4)") ;
                        GET(J)
                 end ;
                 GET(I); PUT(I)
         end
```

4) <Record length> can be used when, for example, you don't want to use the last columns of your input cards because they are irrelevant.

<Record length> must also be used to ensure data are not truncated:

example : FORMAT(3,"*(I7)") provocates a runtime error if more than 18 integers are printed;

FORMAT(3,"120,*(I10)") is always executable .

A10.11. Restrictions and detected errors.

A runtime error 70 occurs when :

- syntax is not respected in <formats> ;
- <number> is equal to 0 or greater than 255 ;
- there exist only <control codes> in a <stream group> ;
- <record length> exceeds 80 if ID = 1, 132 if ID = 3, and 130 if ID = 2 or 4;

- the <format> is too long (ca. 400 bytes in internal representati
- invalid <control codes> according to the device
  (P code and R code when IO = 1 , 2 or 4).
- something wrong with the device (device not ready, not attached,...).

A runtime error 71 occurs when no <format> is 'available'. (in particular when an I/O is requested and no more <alphanumeric code> exits.
example : FORMAT(3,"I4") ; PUT(1,2)).

A runtime error 72 occurs if an invalid device type is specified (IO ne 1, 2, 3, 4)

A runtime error 73 occurs if l>w in a S format code.

A runtime error 74 occurs if :
- a call of GET is elaborated and no <format> is 'available' for IO = 1 or 2 ;
- a call of PUT is elaborated and no <format> is 'available' for IO = 3 or 4.

A runtime error 75 occurs if the mode of the 'next item in the data list' doesn't correspond with the 'next <alphanumeric code>' in the <format> .

example : FORMAT(3,"I4") ; PUT(4.0E0)

A runtime error 76 occurs if the length of a data exceeds 'buffer' capacity (even in a <stream group>).

example :

```
real A , B ;  int I ; [1:100] int C ;
FORMAT(1,"10,2(F7.2)") ; GET(A,B) ;
FORMAT(3,"C150,I4") ; PUT(I) ;
FORMAT(3,"*(I7)") ; PUT(C) ;
```

A runtime error 77 occurs when a virtual "end of file" has been detected on the virtual reader (physical end of file or '*EOF' characters) or on the terminal ('*EOF' characters).

However, see the recovery of runtime errors at Appendix 7.

Other errors, such as invalid characters in input fields are detected by procedures INI,INLI,INR,INLR,OUTI,OUTLI,OUTR,OUTLR,OUTFR,OUTFLR.

example :

```
int I ;
FORMAT(3,"I2"); PUT(999);
 # runtime error 19 occurs #
FORMAT (1,"I3") ; GET(I) ;
 # runtime error 15 occurs
   if the next field in the
   card contains 1E3 #
```

Note that, when a runtime error 70-77 has occured, the <format> is no longer 'available'.