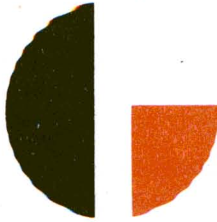


DATA GENERAL
CORPORATION
Southboro,
Massachusetts 01772
(617) 485-9100

PROGRAM

ALGOL User's Manuals:

1. HOW TO PROGRAM IN ALGOL

2. NOVA ALGOL REFERENCE MANUAL

3. REVISED REPORT ON THE ALGORITHMIC LANGUAGE ALGOL 60

ABSTRACT

Data General's ALGOL is a superset of ALGOL 60 with extensions that allow simplified free-form I/O or formatted output, bit manipulation, easy manipulation of character string data, recursive and reentrant procedures, dynamic storage allocation, n-dimensional arrays, multi-precision arithmetic, dynamic type conversion for full mixed-mode capability, and explicit diagnostics.

Copyright (C) Data General Corp., 1970

093-000052-00

INTRODUCTION

ALGOL is a language in which complex mathematical algorithms can be concisely written. The features which differentiate ALGOL from other commonly used languages include recursive procedures, dynamic storage allocation, a modular "block" organization, long variable names, integer or character labels, and a very flexible, generalized arithmetic.

Data General's extended ALGOL compiler for the Nova and Supernova is suitable for business applications and for complex systems programming, as well as research and engineering applications. Extensions provide for the manipulation of character strings, pointer and based variables, and subscripted labels. Data General's ALGOL provides unlimited precision arithmetic, allowing the user to achieve, for example, up to 30 digits of precision.

For business applications, the most attractive features of Data General ALGOL are its character string manipulation, multi-precision arithmetic, file manipulation, and simplified I/O procedures, which provide free-form or formatted output.

Data General ALGOL is a full implementation of ALGOL 60. Recursive procedures are allowed. An array declaration may be any arithmetic expression, including function calls. Integer labels and conditional expressions can be used.

No other mini computers (and few large computer systems) offer a language with the programming features and general applicability of Data General's ALGOL.

CHARACTER STRINGS, implemented as an extended data type to allow easy manipulation of character data. The program may, for example, read in character strings, search for substrings, replace characters, and maintain character string tables efficiently.

MULTI-PRECISION ARITHMETIC, allowing up to 30 decimal digits of precision in integer or floating point calculations.

SIMPLIFIED I/O, using one call for all data types and providing free-form read and write or formatted output according to a "picture" of the output line.

DYNAMIC CONVERSION of parameter type (integer, real, string, Boolean), allowing one program to process data of several types. Expressions of mixed type are allowed.

RECURSIVE AND REENTRANT PROCEDURES, which are particularly important in real-time applications for fast context-switching among programs.

DYNAMIC STORAGE ALLOCATION, freeing the programmer from many details of data layout and storage assignment.

N-DIMENSIONAL ARRAYS, which may be allocated dynamically at run time. Subscripts and array bounds in the array declaration may be any expression including function references, negative numbers, and subscripted variables.

EXTENDED ADDRESSING CAPABILITY, providing more efficient code and more easily understood source language notation. Data General ALGOL draws upon some of the powerful addressing features of PL/I, including based and pointer variables, to permit optional use of the addressing characteristics of the Nova and Supernova computers.

BIT MANIPULATION using logical operators and octal or binary literals. Built-in functions are provided to allow efficient access to data at the bit level.

GENERATION OF EFFICIENT OBJECT CODE and commented assembly language output. Code is optimized for register usage, generation of literals, optimal use of machine instructions, and efficient storage allocation.

OBJECT CODE and RUNTIME COMPATIBILITY with FORTRAN and assembly language to permit referencing not only of external programs and data compiled by the ALGOL compiler, but any object program, whether the source program was written in FORTRAN or assembly language.

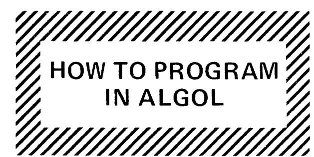
EXPLICIT DIAGNOSTICS to aid debugging at the source level, and compatibility with the Data General symbolic debugger to aid runtime debugging.

HOW TO USE THE ALGOL MANUAL

The ALGOL manual is divided into three separate parts. Part 1 is a tutorial called "HOW TO PROGRAM IN ALGOL". The tutorial presents the basic concepts of ALGOL for programmers unfamiliar with ALGOL or with compiler languages.

Part 2 is a complete description of NOVA ALGOL called the "NOVA ALGOL REFERENCE MANUAL". A combined index to Parts 1 and 2 follows the Appendices to Part 2.

Part 3 is the ALGOL 60 specification, "REVISED REPORT ON THE ALGORITHMIC LANGUAGE ALGOL 60". Programmers familiar with ALGOL need only refer to the specification and those sections of the reference manual dealing with extensions to ALGOL 60.



HOW TO PROGRAM IN ALGOL

CONTENTS

GENERAL PROGRAM ORGANIZATION.. 1

DECLARATIONS.. 3
Why Declarations Are Needed in ALGOL.................................. 3
Size of Storage for Identifiers....................................... 3
Allocation and Release of Storage for Identifiers..................... 4
Data Types.. 4
Arrays.. 5
Lists of Identifiers in Declarations.................................. 7
Local and Global Identifiers.. 7

STATEMENTS..11
Statement Termination...12
Assignment Statement..13
go to Statement...16
if Statement..18
for Statement...20

PROCEDURES..22
Declaring a Procedure...22
Calling a Procedure...23
Returning from a Procedure..23
Identifiers Used in Procedures..24
External Procedures...24
Parameters of Procedures..25
Functions...26
Recursive Procedures..27
I/O Procedures Supplied to the User...................................27
Functions Supplied to the User..31

STRING VARIABLES AND ARRAYS...32

BIT MANIPULATION.. 33

CHANGE OF RADIX... 34

WRITING AN ALGOL PROGRAM.. 35

GENERAL PROGRAM ORGANIZATION

A basic ALGOL program starts with the word *begin* and ends with the word *end*.

```
begin
  .
  .                          ←basic program
  .
 end
```

begin and *end* are written in italics because they are reserved words (called keywords). ALGOL recognizes keywords as having a special meaning; the user cannot change the meaning of keywords or use them for his own program names. The user writes a keyword at the teletypewriter either in all upper case letters or all lower case letters.

A basic program is called a block.

Inside a block are declarations and statements. Declarations list user program names and their characteristics. User program names are called identifiers. Statements show the action the program will take.

Declarations of identifiers must precede their use in statements.

```
begin declarations;
declarations;                ←declarations precede statements
statements;
statements;
end
```

An example of a block, containing declarations and statement is:

```
begin
real pi;
integer k;                      ←declarations
real array R[300], AREA [300];

pi := 3.1416;
for k :=1 step 1 until 150 do   ←statements
AREA [k] := pi×R[k]  ↑2;
end
```

GENERAL PROGRAM ORGANIZATION (continued)

An ALGOL program can be written in free form. This means that a declaration or a statement can be continued from one line to the next and that more than one statement or declaration can be written on a line. For example, the previous program could be written:

```
begin real pi; integer k; real
array R [300
], AREA [300];pi := 3.1416;for
k := 1 step 1 until 150 do
AREA[k] := pi×R[k]↑2; end
```

←but the program is harder to read if it does not have some format.

Separators and Delimiters

Since the end of a line is not a delimiter in ALGOL as it is in the NOVA assembler, other delimiters must be used. A few common ALGOL delimiters are the keywords themselves and the symbols:

;   -usually ends a declaration, statement, or comment.

,   -separates items in a list.

:   -terminates a label definition.
        separates lower and upper bound of array dimensions.

()   -enclose parameters of procedures and built-in functions, precision of arrays, maximum length of strings, and string subscripts.

[ ]   -enclose dimensions of an array and enclose subscripts.

space -separates identifiers that are not otherwise separated such as two keywords together or a keyword followed by an identifier.

Examples of required blank spaces are shown below as triangles. Other blanks are not significant.

```
begin Δ real Δ pi; integer Δ k;
real Δ array Δ R[300], AREA[300];
```

Other delimiters will be introduced later in this manual. The Reference Manual contains a complete list.

DECLARATIONS

Why Declarations Are Needed in ALGOL

When a programmer writes a program for compilation in a high-level language such as ALGOL, he uses several, sometimes a very large number of program variables that are assigned different values during execution.

A declaration tells the ALGOL compiler the name of a program variable, called an identifier. In addition, a declaration shows:

How much storage space the identifier needs.
How and when storage is allocated and released.
What kind of identifier is involved.

Much of this information does not actually appear in most declarations but is given by default. For example:

```
integer k;                    ←declaration of k;
```

tells the compiler:

The identifier is k.
k can have integer values.
Default storage for integers should be used for k.

Size of Storage for Identifiers

The basic NOVA storage unit is a 16-bit word. A word is the default storage unit for an integer value in ALGOL. Other kinds of identifiers that are stored in a single word are:

boolean values	(true or false)
pointers	(not described in this manual; see Reference Manual)

The other kinds of identifiers are those for real (decimal) and complex values, and those for strings of characters. Real values are stored by default in two 16-bit words. Each part of a complex value (real and imaginary) is stored as a real value. There is no default storage unit for a string of characters. The user must tell the compiler how many characters his string can have.

Default storage allocations can be overridden by the programmer for real numbers and integers. If he wants integers greater than

DECLARATIONS (continued)

Size of Storage for Identifiers (continued)

4 digits, he can specify the number of words of storage to be used. He can also specify a larger storage unit or a one-word storage unit for real numbers.

```
integer  (2)k;          ←k is stored in 2 words
real     (1)x;          ←x is stored in 1 word
string   (8)z;          ←z has a maximum of 8 characters
```

To approximate the number of decimal digits of precision that can be stored in a given number of 16-bit words, use the following formulas. n represents the declared precision in words.

integer digits $=5(n-1)+4$ integer range $=\pm 2^{16n}-1$

real digits $=5(n-1)+2$ $10^{-75} \leq$ real range $\leq 10^{78}$

Allocation and Release of Storage for Identifiers

By default, an identifier is allocated storage when the block in which it is declared is entered (*begin* keyword) and the storage is released when the block is terminated (*end* keyword).

A large ALGOL program can be made up of many basic blocks. Some blocks are entered and exited many times. Allocating and releasing storage by block makes more storage available for other identifiers.

However, suppose a programmer wants to enter and exit a block many times during program execution. The block contains a real identifier, R. The programmer wants to enter the block each time, with R having the same value it had when the block was last terminated.

If the programmer declares R with the keyword *own*, R will be stored in a separate area from the other identifiers. In the *own* area, space allocated to identifiers is never released until the entire program terminates.

```
own real R;
```

Data Types

The declaration of a data type tells the compiler the kind of values an identifier can have. The programmer must declare a data type for

DECLARATIONS (continued)

Data Types (continued)

almost all identifiers. He does not declare a data type for labels.

```
integer      x;       ←has values like +15,3,-25
real         y;       ←has values like 3.1416 and -.22266
boolean      z;       ←has value true or false
string  (5)  r;       ←has values like $5.25 or abcde
pointer      p;       ←has an integer value. See Reference Manual.
complex      j;       ←has values like 3.2+2.76i
```

Labels are 'declared' by being used as labels. (However, formal parameters that are replaced by labels are declared *label*. See section on procedures.)

```
100: x :=3;           ←100 is a label on the statement
                       x :=3;
```

Arrays

So far, only identifiers that can have one value at a time have been used. It is possible to declare an array. An array is an identifier of an ordered set of values. Each member of the set is called an array element.

Arrays, like simple identifiers, are declared with a data type and storage characteristics. These apply to each element in the array.

```
integer (2) array Matrix; ←declaration of array, Matrix. Each
                           element in Matrix can have an integer
                           value up to 9 digits long.
```

If you look at the declaration of Matrix, you see that the compiler has no way of knowing how many elements Matrix is supposed to have. While this kind of array declaration is used under circumstances described later, the programmer will usually declare:

How many elements are in the array.

How each element is to be numbered. (This also will determine the order in which values are stored into identifiers.)

DECLARATIONS (continued)

Arrays (continued)

This part of the array declaration is called dimensioning the array. For example:

```
integer array Matrix[25];
```

The single number 25 tells the compiler that Matrix is an array containing 26 elements, numbered:

Matrix[0], Matrix[1],..., Matrix[25]

and values are assigned in that order.

An array can have more than one dimension. In fact, it can have up to 128 dimensions. For example, an array containing real values for the lengths and diameters of pipe might be written with two dimensions as follows:

```
real array pipe[5,5];
```

The declaration tells the compiler that the array, pipe, has 6x6 or 36 elements. The elements are

pipe[0,0], pipe[1,0], pipe[2,0],...,pipe[6,0], pipe[0,1], pipe[1,1], ...,pipe[6,1],...,pipe[5,6], pipe[6,6]

The identifying numbers of each element in the array are called the subscripts of the array. If you look at the elements of array pipe, you will see that the first subscript varies most rapidly. In an array of several dimensions, values are assigned in this way: the first subscript varies most rapidly, then the second subscript, then the third subscript, etc.

If the programmer wishes, he can give an array a different starting number from zero. For example, array pipe could have been written:

```
real array pipe[-5:0,1:6]
```

Pipe still has 36 array elements but now they are numbered:

pipe[-5,1], pipe[-4,1],..., pipe[-1,6], pipe[0,6]

DECLARATIONS (continued)

Arrays (continued)

The first number of each dimension gives the lower bound of the dimension; the second number gives the upper bound. The lower bound must be a smaller integer than the upper bound. Besides integer and real arrays, arrays of strings can be declared. The maximum length of each element of a string array must be declared, as there is no default number of characters. For example:

```
begin string(8) array ID[9,9];
```

ID is declared as a two dimensional 10x10 array of strings. The maximum length of each string is eight characters.

Variable strings are an extension to ALGOL. Some of the ways in which they can be used are discussed later.

Lists of Identifiers in Declarations

The programmer does not have to write a separate declaration for each and every program variable. Quite often a number of program variables have the same data types and storage characteristics. In this case, the programmer can write one declaration, listing all the identifiers.

```
begin integer i,i1,i2,i3;
real x,y,z;
real array M[5,5], z[8,8];
```

Local and Global Identifiers

The block structure of ALGOL permits blocks within other blocks. In the following diagram, three blocks are shown. The blocks labeled B2 and B3 are inside the block labeled B1.

DECLARATIONS (continued)

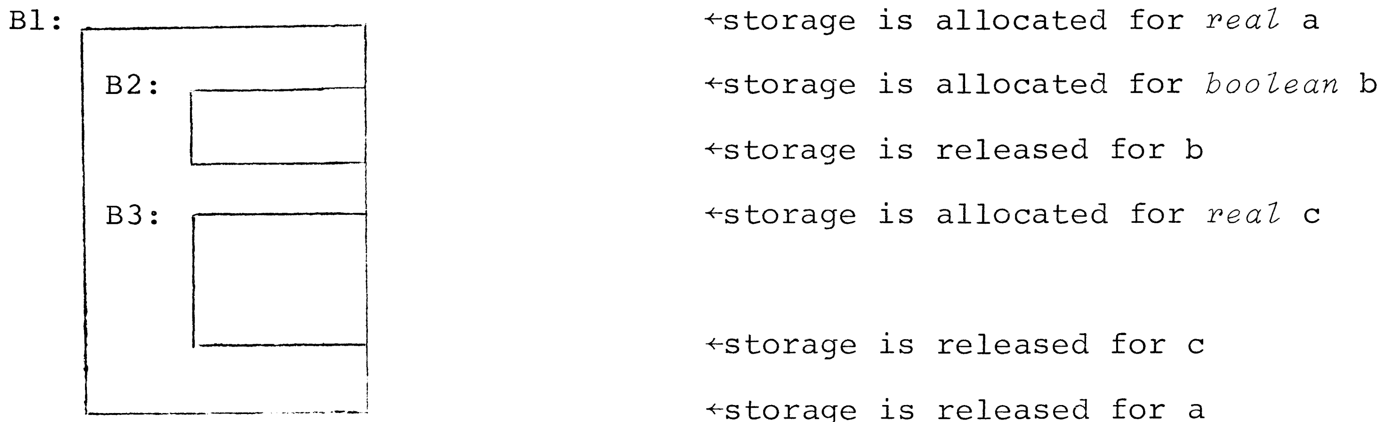
Local and Global Identifiers (continued)

```
B1: ┌begin real a;                  ←a is declared in block B1
    │
    │ B2:┌begin boolean b;           ←b is declared in block B2
    │    │
    │    └end B2                     ←B2 ends.  Note that end can
    │                                 be followed by a string of
    │                                 characters, in this case the
    │                                 block identifier.
    │
    │ B3:┌begin real c;              ←c is declared in block B3
    │    │
    │    └end B3                     ←B3 ends.
    │
    └end B1                          ←B1 ends.
```

Since B2 and B3 are both within block B1, any identifier declared in B1, such as *real* a, is defined for blocks B2 and B3.

Identifier a is said to be local to block B1 (the block in which it is declared) and global to blocks B2 and B3 (the blocks in which it is defined).

Identifier b is local to block B2 and identifier c is local to block B3. Elsewhere, both these identifiers are undefined. Why this is so can be seen in the following diagram of the three blocks.



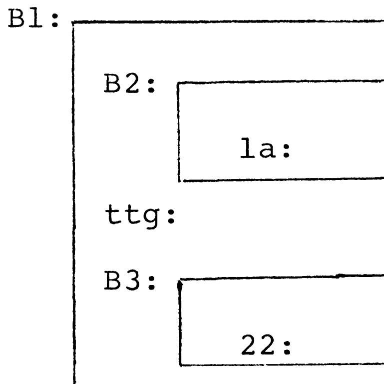
DECLARATIONS (continued)

Local and Global Identifiers (continued)

Labels are declared by their appearance as labels within a given block. For example, the blocks B1, B2, and B3 might each contain labeled statements.

```
B1: begin real  a;
    B2: begin boolean  b;
        la:- - -;                 ←la is a label local to
                                   block B2.
        end B2
        ttg:- - -;                ←ttg is a label local to
                                   B1 and global to B2 and
    B3: begin real  c;             B3.
        22:- - -;                 ←22 is a label local to
                                   B3.
        end B3
    end B1
```

Like declared identifiers b and c, labels la and 22 are undefined except in their own blocks. Note, however, that the labels of the blocks B2 and B3 are outside of the blocks they label and are local to block B1 and global to blocks B2 and B3, as shown in the following diagram:



Even though a label does not appear in a declaration, it is a data type and is stored in one word; if an identifier is used as a label, it cannot be used as any other type of datum.

DECLARATIONS (continued)

Local and Global Identifiers (continued)

Arrays can be declared with variable dimension bounds such as

```
real array z[i,j];
```

where the bounds of array z and 0 to i and 0 to j. However, the appearance of variable dimensions in an array declaration constitutes a use of identifiers i and identifiers must be declared and defined before the block containing the array declaration is entered. Thus, i and j must be global to the block containing the declaration of z. For example, the following is legal:

```
B: begin integer i,j;              ←i and j are declared and their
        .                           values defined in block B.
        .
        .
        i :=50;                    ←statements that assign
        j :=100;                    values to i and j.
        .
        .
        .
   C: begin real array A[i,j];     ←i and j can then be used as
        .                           dimensions of array A in block C.
        .
        .
      end C
        .
        .
        .
   end B
```

However, the following is illegal:

```
   B1: begin integer i,j;
             real array A[i,j];    ←i and j are used before their
                                    values are defined.

   end B1
```

A later section describes procedures and the formal parameters of procedures. Formal parameters are not allocated storage as are actual program variables and labels; therefore, the rules of declaration and definition before use do not apply to formal parameters.

STATEMENTS

Statements are programming instructions. They indicate how operations are to be performed using the declared identifiers.

ALGOL statements are very flexible so that programmers unfamiliar with ALGOL can use short, simple statements. Experienced ALGOL programmers, however, can nest statements within other statements. In fact, an entire block may be treated as a single statement.

Some examples of simple statements are:

```
a := b+1.0;
```

←Assignment. b+1.0 is evaluated and placed in location a.

```
go to Lb_13;
```

←Unconditional transfer to the statement labeled Lb_13.

```
if bool then go to B;
c :=c/d;
```

←Conditional transfer.

bool is a Boolean variable. If bool has the value *true*, a transfer is made to the statement labeled B. If bool is *false*, the assignment statement is executed.

```
tag2:;
```

←Dummy statement providing a label to which to transfer.

```
for i :=0,2,25 do
x[i] := y[i]+i;
```

←*for* statement.

The *for* statement causes a loop. The variable i is assigned the first value (0) of the list 0,2,25, and the assignment statement is executed. Then i is assigned the second value (2) and the assignment statement is executed, etc.

```
proc23(x,y,z);
```

←procedure call

A call to a procedure named proc23 is made from the current block.

STATEMENTS (continued)

comment: Comments contain explanatory information;

ALGOL comments are written as statements, beginning with the keyword *comment* and ending at the first semicolon.

Often, a programmer wants a group of statements to be treated as a single statement. A common example is a group of statements following a *for* statement, where the programmer wants the loop to include the group of statements. He can use the keywords *begin* and *end* to "block" his statements.

```
for p :=5,10,15,20
do begin
     A[p] :=p↑2;
     B[p] :=A[p] -x;
     C[p] :=B[p] +A[p];
end
```

The three assignment statements will be executed for each value of p.

Statement Termination

Statements shown previously have generally been terminated by a semicolon. However, statements may be terminated in some instances by the keyword *end* or the keyword *else*. For example, the previous compound statement could be written:

```
for p :=5,10,15,20
    do begin
        A[p] :=p↑2;
        B[p] :=A[p] -x;
        C[p] :=B[p] +A[p]              ←end terminates this statement
      end
```

The keyword *else* can terminate a statement in a conditional clause.

```
if x=0 then go to LABLAA else          ←else terminates go to LABLAA
if x>0 then y :=x
else                                   ←else terminates y :=x
x :=x + 1;
```

STATEMENTS (continued)

Statement Termination (continued)

The keyword *end* can be followed by a string of characters, for example:

```
end of block 25
```

Because statements are terminated by keywords *end* or *else* or by semicolon (;) delimiters, the string of characters after an *end* cannot include these characters.

Assignment Statement

The basic statement is the assignment statement that permits the value of an expression to be stored in location represented by an identifier.

```
variable :=expression;
          ↑
      assignment
        symbol
```

```
begin real b,c; integer a; boolean boo;
        .
        .
        .
a :=0;          ←assignment of constants to variable
b :=c :=2.5;     locations.  Note that 2.5 is assigned
boo :=true;      both to location b and location c.
```

```
a :=a+2;        ←simple expressions.
b :=c↑3;         ↑   shows exponentiation.
boo :=┌boo;      ┌   means logical not.
```

STATEMENTS (continued)

Assignment Statement (continued)

ALGOL expressions can be relatively simple as shown above or can represent highly complex processes. A few more simple expressions might be

```
v
z+4
(-b+sqrt (d) )/2/a                   /shows division.
d+abs (w[0] -y×w[1] -sin (x/2)       ×shows multiplication.
w[k[i]]                               subscripts can be nested to
                                      any depth.
```

Note the terms abs, sin, and sqrt in the expressions. These are references to functions, and the parenthesized expressions following the function reference are the actual parameters passed to the function when it is referenced. Functions and how they are referenced are described later in a section on procedures.

The variable on the lefthand side of the assignment and the expression on the right must have compatible data types. Each variable type can be assigned an expression of the same data type as in:

```
begin real x,y;
integer i,j; pointer p;
boolean b,c; string (8) char;

     i :=j-4;                         ←integer to integer

     x :=x/y×3.5;                     ←real to real

     b :=¬c;                          ←boolean to boolean

     char :="$25.10";                 ←string to string

     p :=address(y);                  ←address is a pointer function
```

STATEMENTS (continued)

Assignment Statement (continued)

In addition, certain conversions are possible.

```
begin integer i,j; boolean b,c;
     .
     .
     .
   i :=b∧c;                          ←boolean to integer
     .                               ∧is logical and
     .
     .
c :=j;                               ←integer to boolean
```

If b∧c evaluates to *true* (1), integer i will contain 1 in every bit position and if the expression is *false*, i will be a word of all zeroes.

In integer to boolean conversion, the integer expression (j in this case) is evaluated. c will be assigned the value *false* if j contains all zeroes and will be *true* in every other case.

```
begin integer i,j; pointer p;
     .
     .
     .
   j :=p+5;                          ←pointer to integer
     .
     .
     .
   p :=i;                            ←integer to pointer
```

A pointer is one word long and contains a memory location (integer). Therefore, integer to pointer and pointer to integer conversion is permissible with the limitation that the integer must be one word long (default precision).

STATEMENTS (continued)

Assignment Statement (continued)

```
begin integer i,j; real x;
     .
     .
     .
   x :=i↑2;                         ←integer to real
     .
     .
     .
   j :=x/3;                         ←real to integer
```

An integer expression is converted to real by evaluating it and placing the decimal point after the last digit. A real expression is converted to integer by evaluating it and using a function called the entier function. The entier function selects the highest integer value not greater than the real value.

```
begin string (5) char; integer i;
     .
     .
     .
   i :=25;
     .
     .
     .
   char :=i-4;                      ←integer to string
```

The integer is evaluated and then assigned to the string variable as a character string, in this case 21.

All the expressions described in this section are simple expressions. There are also conditional expressions which may be used in assignment statements. Conditional expressions are described in the Reference Manual.

go to Statement

A *go to* statement transfers control to another statement in the program. The keywords *go to* are followed by a label or an expression that evaluates to a label. The expression can be a subscripted label variable or a switch identifier.

STATEMENTS (continued)

go to Statement (continued)

Labels are either identifiers (alphnumeric characters beginning with a letter) or unsigned integers.

```
tagl: x :=x+1.0                    ←identifier label
      .
      .
      .
go to tagl;
      .
      .
      .
go to 10;
      .
      .
      .
10: y6 :=y×x;                      ←integer label
```

A subscripted label variable in a *go to* statement evaluates to a subscripted label. Labels can have a single subscript.

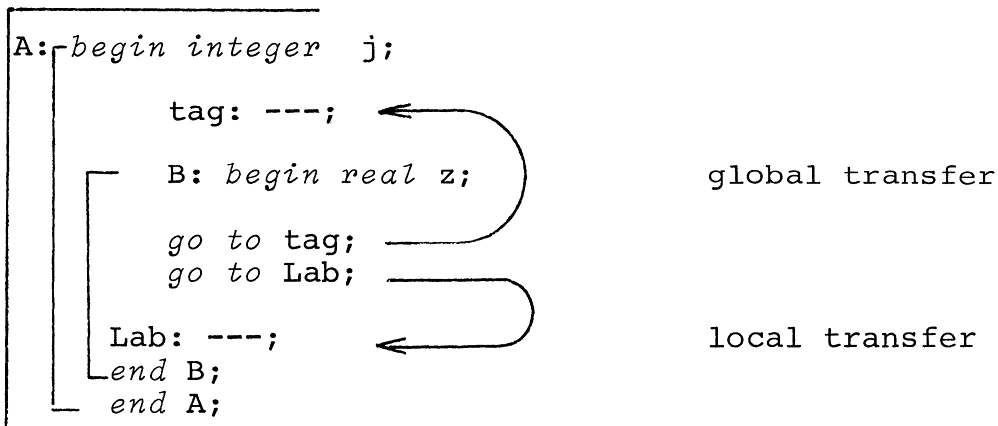
```
tage[1] :x :=x+pi/4;
           .
           .
           .
tage[2] :x :=+pi/2;                 subscripted labels
           .
           .
           .
tage[3] :x :=x+pi;
           .
           .
           .
        go to tage[I];             ←I evaluates to 1,2,or3
```

Switch designators are described in the Reference Manual. They also appear as subscripted expressions to be evaluated in the *go to* statement. In general, switches are less efficient than subscripted labels.

STATEMENTS (continued)

go to Statement (continued)

Because of the way identifier storage is allocated and de-allocated by block, a statement must transfer control within the block or to an identifier global to the block.

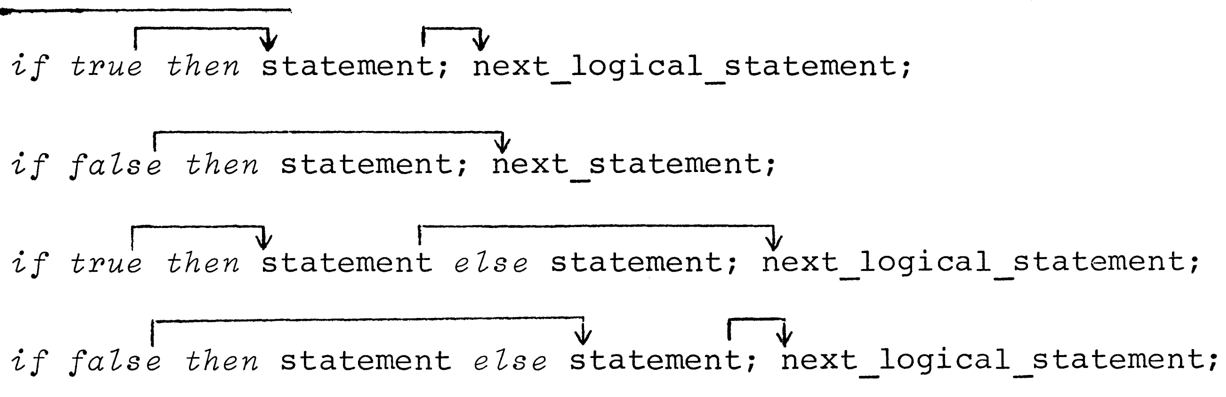


if Statement

if statements use a truth value as a switch to determine transfer of control. There are two formats.

```
if boolean_expression then unconditional_statement;

if boolean_expression then unconditional_statement else statement;
```

If the boolean expression evaluates to *true* the *then* statement is executed; otherwise, the *then* statement is skipped. The arrows in the example below show how control is passed.



STATEMENTS (continued)

if Statement (continued)

Boolean expressions and the logical and relational operators used in forming them are described in the Reference Manual, which should be consulted if you are not familiar with Boolean logic. Briefly, a boolean expression consists of

a+b=c	simple arithmetic expressions (a+b and c) are used with relational operators (= < ≦ = ≧ > ≠)
boo ∧loob	boolean expressions (boo and loob must be declared *boolean*) used with logical operators ┌(*not*) ∨(*or*) ∧(*and*) ⊃(*imp*) ≡(*eqv*) ⊕(*xor*)
a+b=c∨boo ∧loob	a combination of the above two boolean expressions

The *then* statement can be any statement or set of statements as long as it doesn't contain another *if* statement.

```
if a≠b then a :=b;                  }
     .                              }  simple then statements
     .                              }
     .                              }
if c>d then go to 25;               }
     .
     .
     .
if e<5 then begin x :=y :=x↑2;      }
                  y :=y+25.26;      }  blocked then statements
                  go to 30 end      }
```

The *else* clause can be an *if* statement. This means that a series of switches can be set up. For example, the previous statements could be rewritten

```
if a≠b then a :=b else
if c>d then go to 25 else
if e<5 then begin
            x :=y :=x↑2;
            y :=y+25.26;
            go to 30 end
```

STATEMENTS (continued)

if Statement (continued)

Simple expressions were discussed in the section on the assignment statement. The sequence

```
if boolean_expression then...
```

is a conditional expression and can appear anywhere a simple expression can be used, except the following the keyword *then*. Conditional expressions follow the rules for data typing. See the Reference Manual for information on conditional expressions.

for Statement

The *for* statement allows a given statement or statements to be executed repetitively with a controlled variable set to different values. The statement or statements are executed as many times as there are values for the controlled variable. The statement format is:

```
for controlled_variable := list_of_arithmetic_values
do statement(s)
```

At its simplest, the list can contain only values as in:

```
for j :=1, 25, 350, 4, -6 do A[i,j] :=B[j];
```

However, the list can contain variables and expressions.

STATEMENTS (continued)

for Statement (continued)

```
for j :=1, a+3, x/y, if x≠y then 25 else -6 do A[i,j] :=B[j];
```

In addition, a list item can contain either the keyword *while* or the keywords *step* and *until*. A *while* clause would be:

```
for x :=y/2 while y≠z do...
```

The keyword *while* is followed by a boolean expression. The statement following *do* executes as long as the boolean expression is *true*.

A *step-until* clause would be

```
for a :=1 step  2 until  101 do...
        ↑        ↑         ↑

    initial  incre-     final
    value    ment       value
```

The list item is equivalent to the simple list: 1,3,5,...,101. The initial, incremental and final values can be any expression or value. Some examples of *for* statements are:

```
for i :=0.1 step -0.01 until .005
    do x :=i×ln(x);               ←ln(x) is the natural logarithm
                                   function

for j :=1 step 1 until 100 do
      A[i] :=B[i]-C[i];


for k :-0, x-y while x>y,-5 do
    begin A[k,2] :=B[k],          ←compound statement following begin
          C[k] :=A[k,2] +B[k]/x;  ←both assignment statements are
          end                      executed as part of the loop
```

PROCEDURES

Procedures are basic ALGOL programs that are called for execution. Begin blocks can be entered by sequential execution of statements. Procedures are only entered when they are called.

Declaring a Procedure

The format of a procedure declaration consists of a heading and the text or body of the procedure. The body of a procedure can be a single statement, a group of statements delimited by *begin* and *end* as described on page 12, or a block containing declarations and statements.

At a minimum, the heading of a procedure must contain the word *procedure*, followed by the procedure identifier. In addition, the heading may contain additional information about the procedure, described later in this section.

The procedure identifier follows the word procedure in the declaration. Then the text of the ALGOL procedure is written.

```
Z:   begin


procedure ZERODIV;          ←procedure ZERODIV is declared in block Z.
      .                     ←statement containing procedure body
      .
      .
```

Rules that apply to other identifiers apply to procedures as well. A procedure must be declared before it is used (called). It must be declared in the block in which it is called unless, like some identifiers, it is an *external* procedure.

Assume that ZERODIV is a program that is used to prevent errors resulting from division by zero. ZERODIV sets up the following algorithm:

given: c :=a/b the following results are produced:

a value	b value	resulting c value
any	≠ 0	a/b
>0	0	999999
=0	0	0
<0	0	-999999

PROCEDURES (continued)

The full declaration of ZERODIV could then be:

```
Z: begin
     .
     .
     .
procedure ZERODIV;
          if b≠0 then c :=a/b else
          if a=0 then c :=0 else
          if a<0 then c :=-999999 else
             c := +999999;
```

Calling a Procedure

A procedure is called by writing its name as a statement.

```
Z: begin real array  R[10,10], z[10,10,10], Y[10,10,10];
     .
     .
     .
procedure ZERODIV;
     if b≠0 then c :=a/b else
     if a=0 then c :=0 else
     if a<0 then c :=-999999 else
        c := +999999;

          .
          .
          .
     a :=R[i,j];             ←Assign array elements to dividend and divisor.
     b :=z[i,j,k];
     ZERODIV;                ←Call ZERODIV.
     Y[i,j,k] :=c;           ←Put result in proper location.
```

Returning from a Procedure

When a procedure is called, it executes until the end of the procedure is reached. The procedure then returns control to the statement immediately following the calling statement. In the ZERODIV example above, control returns to the assignment statement.

PROCEDURES (continued)

Returning from a Procedure (continued)

```
Y[i,j,k] :=c;
```

Identifiers Used in Procedures

ZERODIV is a block inside the block named Z. Both Z and ZERODIV use the identifiers a,b, and c. If a,b, and c are declared within ZERODIV, they will be undefined in block Z by the rules of block structure. Therefore, a,b, and c are declared in block Z.

```
Z:  begin real a,b,c;
    procedure ZERODIV;
        .
        .
        .
```

There are identifiers that are used only in a given procedure and they can be declared in the procedure.

External Procedures

An ALGOL procedure declaration can be compiled separately from any enclosing block. It can then be used as an external procedure by many programs. Assume that the procedure ZERODIV was compiled separately from any other block. Now, any block can call ZERODIV if the block has a declaration of ZERODIV as *external*.

```
begin
     external procedure ZERODIV;
     .
     .
     .
ZERODIV;                              ←call to ZERODIV
     .
     .
     .
```

PROCEDURES (continued)

Parameters of Procedures

The previous example showing ZERODIV as an *external* procedure raises the problem of identifiers a,b, and c once more. Must they be declared in each and every program that wants to call ZERODIV? ALGOL solves this problem by allowing the user to put dummy identifiers, called formal parameters into a procedure declaration. Then, the procedure can be called with real identifiers, called actual parameters.

With formal parameters, the declaration of ZERODIV could be:

```
procedure ZERODIV(a,b,c);          ←a,b, and c are formal parameters
real a,b,c;                        ←a,b, and c are declared.
if b≠0 then c :=a/b else
if a=0 then c :=0 else
if a<0 then c :=-999999 else
   c :=999999;
```

A parenthesized list of formal parameters follows the procedure identifier. These formal parameters will be replaced when the procedure is called.

The formal parameters must have data types specified. In the example, a,b, and c are specified as *real*.

If the body of the procedure is a block, formal parameters must be specified in the procedure heading, not in the block. If parameters are declared inside the block that is the procedure body, they will be undefined in the procedure heading.

Assume the same block used previously to call ZERODIV now wishes to call it to obtain a value for Y[i,j,k].

```
begin  real  array  R[10,10], z[10,10,10], Y[10,10,10];
     .
     .
     .
  ZERODIV          (R[i,j], z[i,j,k], Y[i,j,k]);  ←Call to ZERODIV
```

When ZERODIV is called, array element R[i,j] replaces a, z[i,j,k] replaces b and Y[i,j,k] replaces c. There is no need to assign the values in the calling block. The assignment is made when the actual parameters are passed in the call.

The rules governing formal and actual parameters are given in the Reference Manual.

PROCEDURES (continued)

Parameters of Procedures (continued)

cannot replace a simple variable and an array cannot replace a simple variable. Some legal substitutions are:

```
begin real x,y,z; external procedure sum;                 \
procedure XX(a,b,c,d,e,f);          \                      |
     real a,b,c; boolean d;         |                      |
label e; real procedure f;          |   Procedure          |
begin .  \                          >   Declaration        |
      .  |  procedure body          |                      |
      .  |  (statement              |                      |
end      /     or block)            /                      |  Calling
      .                                                    >  Block
      .                                                    |
      .                                                    |
XX(x,y,z, true, Exit, sum);            <-call              |
      .                                                    |
      .                                                    |
      .                                                    |
Exit:---                                                   |
end                                                        /
```

Because formal parameters are only dummy identifiers, their declarations are not as restrictive as that of real identifiers. Note in the example that a label can be declared. Also, it is often useful to leave a parameter declaration somewhat vague to allow a larger number of possible replacements. For example an array formal parameter could be declared without dimensions.

Functions

A function is a procedure which, upon execution, results in a value. In fact, at some point in the function, an assignment statement assigns a value to the function identifier.

Since a function represents a value, it must have a data type. A data type is included in the declaration of a function.

```
real procedure arctanh (x);
real x;
arctanh := 0.5 ln((1+x)/(1-x));     <-value is assigned to arctanh
```

PROCEDURES (continued)

Functions (continued)

Since a function represents some value, a function call is part of an assignment statement or other statement:

```
      .
      .
      .
real procedure arctanh (x);
      .
      .
      .
z :=z×arctanh(y);  ←function call to arctanh with actual parameter y
```

When execution of arctanh is complete, the value of arctanh replaces the call in the assignment statement.

A function has one of the ALGOL data types: *integer, real, string, complex, boolean, pointer,* or *label.* (A label can be specified as a function type.)

Recursive Procedures

ALGOL permits recursive procedures. A procedure is recursive if it calls itself. An example is factorial computation.

```
integer procedure factorial(I);
integer I;                                        }Declaration of
factorial := if I=0 then 1                        }integer function,
else factorial (I-1)×I;    ←factorial calls itself}factorial.
```

I/O Procedures Supplied to the User

ALGOL does not provide for I/O operations. Five externally compiled procedures are supplied with NOVA ALGOL to handle user I/O.

Before proceeding with I/O operations, the user must open a file for input or output. The "file" can be a data file in secondary storage or an I/O device.

To open the file the user writes the call:

```
open(number,string);
```

PROCEDURES (continued)

I/O Procedures Supplied to the User (continued)

The number is one of 8 channels (0 to 7) that can be associated with a given file and the string is the name of the file.

```
open (2, "myfile");

open (3, "$TTO");          ←$TTO is the teletypewriter on output
```

Once a file has been opened, data can be read or written from it. The read and write calls are:

```
read(number,list);

write(number,list);
```

The number is again the channel number associated with the file. The list is a list of variables, expressions, and string constants to be read or written from the file.

```
open (2, "myfile");

write (2, a,b,c,d, "timings follow:", ArraY);
```

In the example, the user opens myfile and associates channel 2 with it. He then requests that certain variables a,b,c,d be written to the file. They will be written out according to the way they are formatted in the file and their data type; the user does not have to format them. The user then inserts a string constant 'timings follow:'. After this, ArraY, which is presumed to be an array of timing information, will be written to the file.

When I/O operation for a given file is completed, the file must be closed. This insures proper updating of the file and releases the association between the file and the channel number. The format for the call is:

```
close(number)
```

PROCEDURES (continued)

I/O Procedures Supplied to the User (continued)

A fifth I/O routine allows the user to generate data output in a large number of possible formats. The call is:

```
output(number,format,variable_list);
```

The number is the channel number. The variable list is a list of variables, expressions, and string constants to be output. The list can also include carriage control functions tab and page.

Tab and page are functions supplied to the user. The page function causes a form feed. The tab function has a parameter that is the number of the character position to which the teletypewriter will tabulate. Function page should always be enclosed in parentheses in the output list.

How the user writes his format parameter determines the format of the output values. The format parameter is enclosed in either accent marks or double quotation marks. If the user puts text in the format parameter, the text will be output exactly as written.

```
output (2, "              RESULTS OBTAINED ARE:");

            RESULTS OBTAINED ARE:          ←output
```

The user can also set up a field format for his data, using a # for each character position.

```
output (2, "RESULTS OBTAINED ARE:  ####",a);

RESULTS OBTAINED ARE:     345      ←datum in location a is written in
                                    the given format
```

If the output number is smaller than the field format, it is right justified with leading blanks.

PROCEDURES (continued)

I/O Procedures Supplied to the User (continued)

A decimal point can be used in a field format. Assume variables have the following values: x=-456.78, y=999.123, z=.08

```
output (2, "#####.# ",x,y,z);

  -456.8   999.1      .1
```

Signs + or - can be used in a field format. Without the sign, as previously shown, only negative values are output with a sign, and the minus sign requires a field format position.

If a plus sign is given, both positive and negative values are output with signs. If a minus sign is given, only negative values are output with signs. However, in both cases, the sign does not require a field format position.

```
output (2, "-####.# "b,c);

  -4567.2  5858.0           ←both positive and negative numbers can
                             have four digits before the decimal point
```

Character strings are output in the same format as decimal numbers, using # for each character position. The character string output can be from a variable in a file or can appear as a literal in the list of variables of the output statement.

```
output (2, "##########", i,j, "Price");   ←i and j are variables and
                                           Price is a literal

Item No.   Stock No. Price                ←possible output
```

Character strings are left justified in the field format with following blanks.

If the character string is longer than the field format, the entire string will be written.

```
output (2, "###", "ADDRESS");

ADDRESS
```

PROCEDURES (continued)

I/O Procedures Supplied to the User (continued)

In I/O procedure calls - read, write, and output - an array identifier in the output list causes all elements of the array to be transferred in order. In the next example, assume A is an array of ten integer elements.

```
output  (2,  "####", A);

 345 7777  567   23 4567  890  230   46 9012 7564  ←possible output
```

Carriage control functions, tab and page, can be inserted in the variable list to set up special formats.

```
output  (2,  "#####", "STOCK ITEM", tab (5), A);

STOCK ITEM
       345
      7777
       567
        23
      4567
       890
       230
        46
      9012
      7564
```

In the example above, the string constant STOCK ITEM extends beyond tab position 5. The tab function will cause a line feed before tabulating to character position 5. Each element of array A is then right justified in the field beginning at tab position 5.

The Reference Manual contains additional examples of how a user can format output. It includes examples of how a user can prepare a table of values using *for* loops and output calls.

Functions Supplied to the User

ALGOL has certain standard arithmetic functions that are supplied to the user such as those for taking a sine, cosine or square root (sin, cos, sqrt). In addition NOVA ALGOL has a number of additional functions to permit the user to manipulate character strings, bit strings, and for using pointer variables. Some of these special

PROCEDURES (continued)

Functions Supplied to the User (continued)

functions will be discussed in the sections following and others are described in the Reference Manual.

STRING VARIABLES AND ARRAYS

String variables and arrays are an extension to the ALGOL language that allows manipulation of character strings. A complete string or part of a string may be referenced:

```
string (9) a,b;
     .
     .
     .
a := `xxxyyyzzz';
a :=a(3,6)                    ←b :=`xyyy'; or character positions
                               3 through 6.
```

In a similar way, strings and string arrays can be manipulated. for example:

```
string (9) a;
string (2) array c[1:8];
     .
     .
     .
a :=`xxxyyyzzz';

for i :=1 step 1 until 8 do
c[i] := a (i,i+1)             ←note use of subscript brackets
                               and substring brackets
```

When the *for* statement is executed, the contents of the elements of array c will be:

c[1] :="xx"; c[2] :="xx"; c[3] :="xy"; c[4] :="yy"; c[5] :="yy"; c[6] :="yz"; c[7] :="zz"; c[8] :="zz";

Concatenation of strings can be handled by the length built-in function, which returns the integer length of a string as its value.

```
string(10) a,b;
a :=`xxx';
b :=`yyy';
a (length (a)+1) :=b;         ←string variable b is concatenated
                               with a to form `xxxyyy'.
```

The lefthand side of the final assignment evaluates to a(4) :=b; which means "start putting the contents of b into a, beginning at the fourth character."

STRING VARIABLES AND ARRAYS (continued)

The index built-in function returns a value that represents the character position of a character of the string.

```
string (10) a; integer b;
a :='xyzzz';
b :=index (a,'y');              ←statement is equivalent to b :=2;
```

Coding of the index function shows how string variables and the length built-in function can be used in programming.

```
integer procedure index (a,b); string a,b;
begin integer i;
     i :=0;
     for i :=1 step 1 until length (a) do
     if a(i,length (b)+i-1)=b then go to done;
done:     index:=i;
     end
```

BIT MANIPULATION

An extension to ALGOL allows the programmer to perform bit manipulation in the higher-level language, comparable to assembly language coding. This is possible by use of the built-in shift and rotate functions and by use of binary and octal literals that have been added to the syntax. Bit manipulation cannot be performed upon *real* variables, which are represented in floating point.

A simple example of the shift function is

```
x :=shift (y,-4);               ←left shift variable y four bits and
                                 assign the value to x.
```

The rotate function is similar

```
x :=rotate (y,+2);              ←right rotate variable y two bits
                                 and assign the value to x.
```

BIT MANIPULATION (continued)

If the programmer wants to select bits from an integer, he can use logical operations and binary or octal literals to mask unwanted bits. For example, suppose x is an integer containing a 3-bit index into an array in bits 5,6,7.

```
0           5 6 7                    15
[ |         | | | |                 | ]
             \___/
             index
```

The variable i can be set to contain the index as follows:

```
i :=shift(x,-8) and 111r2;

        or

i :=shift(x,-8) and 7r8;
```

where:	shift(x,-8)	causes	xxxxxxxxxxxxxxxx (index)
and either 111r2 or 7r8		causes	0000000000000xxx (index)

New settings for bits of a given variable can also be made using logical operations and/or the built-in functions.

CHANGING A RADIX

NOVA ALGOL permits the user to set any radix up to and including 10, as shown for base 2 and base 8 in the section on bit manipulation. Simply follow the literal with the letter r and the radix:

.1r3	- base 3
1.3E9r4	- base 4. The exponent is 4^{9}
101E-10R2	- base 2. The exponent is 2^{-10}

WRITING AN ALGOL PROGRAM

The steps to follow in writing an ALGOL program are:

1. Study the problem. Can it be broken into several algorithms? Can you further generalize the algorithms for repetitive use? The first decisions are how to structure the problem - nested blocks, separately compiled procedures, etc.

2. When you decide upon the structure of your program you should decide what identifiers - variables, arrays, parameters, etc. - need to be declared in each block. Declaration of identifiers may be new to some programmers. It is essential to ALGOL programming.

   Be sure the data types you select are suitable not only for data storage but also as to compatibility of formal and actual parameters and variables that will be used together in expressions.

   Decide on the precision of integer and real data that you will need.

3. When the declarations have been written, the statements that implement the program can be written. Be sure to label statements you will transfer to and to write comments. Comments will help both you and other programmers.

4. Before attempting compilation, make a source-program debugging check. Have you put in the proper delimiters, blank spaces, and spelled the identifiers correctly?

5. When you attempt compilation, check the error messages carefully against your source program and make the necessary changes.

6. When you get your first ALGOL programs to compile, chances are they will not be very efficient. Check the compiled code carefully. Have you made full use of supplied functions, nesting of procedures, and external procedures? Have you used bit manipulation facilities? Experiment with your source program and see if you can improve the coding.

7. As you become more proficient in writing ALGOL programs, try to use the additional facilities described in the Reference Manual such as pointers and based variables. These facilities for sophisticated programmers such as systems programmers will also improve your coding efficiency.

NOVA ALGOL REFERENCE MANUAL

NOVA ALGOL REFERENCE MANUAL

CONTENTS

IDENTIFIERS AND KEYWORDS....................1- 1
  Keywords....................1- 1
  Function Keywords....................1- 1

SCOPE OF IDENTIFIERS....................2- 1
  Variables, Arrays, Switches and Procedures....................2- 1
  Labels....................2- 1
  Parameters....................2- 2
  Storage and Scope....................2- 2
  Scope and Blocks....................2- 3

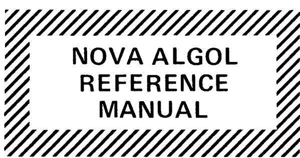
BLOCKS....................3- 1

DELIMITERS....................4- 1
  Separators....................4- 2
  Brackets....................4- 3
  Arithmetic Operators and Numbers....................4- 4
  Boolean Operators and Values....................4- 7
  Rules of Arithmetic and Boolean Expression Evaluation....................4- 8
  Bit Operations....................4- 9

EXPRESSIONS....................5- 1
  Arithmetic Expressions....................5- 1
  Boolean Expressions....................5- 4
  Pointer Expressions....................5- 5
  Designational Expressions....................5- 6

STATEMENTS....................6- 1
  Assignment Statement....................6- 3
  for Statement....................6- 7
  go to Statement....................6-11
  if Statement....................6-12

IDENTIFIER DECLARATION AND MANIPULATION....................7- 1
  Data Types....................7- 2
  Arrays....................7- 3
  Character String Variables....................7- 6
  Pointers and the *based* Declarator....................7- 9
  Labels....................7-13
  Switches....................7-15
  own Declarator....................7-16
  external Declarator....................7-16
  Literals....................7-17



Procedures.. 8-1
Procedure Declarations.................................... 8-1
External Procedures....................................... 8-3
Procedure Calls... 8-3
Calling a Procedure by Name and by Value.................. 8-5
Formal and Actual Parameters.............................. 8-6
Specificators of Formal Parameters........................ 8-8
Procedure Coding Example.................................. 8-8

BUILT-IN FUNCTIONS.. 9-1
Mathematical Functions.................................... 9-1
Transfer Function... 9-2
Address Function.. 9-2
Length Function... 9-3
Index Function.. 9-3
Shift Function.. 9-4
Rotate Function... 9-4
Tab Function.. 9-5
Page Function... 9-5

I/O RUN-TIME PROCEDURES...10-1
Open a File...10-1
Read a File...10-2
Write a File..10-2
Close a File..10-3
Formatted Output..10-4

ALGOL ERROR MESSAGES..11-1

EXTENSIONS TO AND LIMITATIONS OF STANDARD ALGOL..................12-1

APPENDIX A - CALLS AND RETURNS

APPENDIX B - RUN-TIME ROUTINES

APPENDIX C - LOADING THE ALGOL COMPILER

APPENDIX D - EXAMPLE OF ALGOL GENERATED CODE

IDENTIFIERS AND KEYWORDS

An identifier is a string of one to 32 letters, digits, and underscore symbols (_) that must begin with a letter. Identifiers are names assigned by the programmer to variables and other program entities. Upper or lower case letters may be used. No blank spaces are permitted.

Examples:

a	get_symbol	Routine_2
A25	Aa	omega

Identifiers serve to identify simple variables, arrays, labels, switches, procedures, and pointers.

Keywords

Certain keywords are completely reserved in ALGOL. They are:

and	*do*	*go to*	*own*	*switch*
array	*else*	*if*	*pointer*	*then*
based	*end*	*imp*	*procedure*	*true*
begin	*eqv*	*integer*	*real*	*until*
boolean	*external*	*label*	*step*	*value*
comment	*false*	*not*	*string*	*while*
complex	*for*	*or*		*xor*

Keywords must be written in all upper case or all lower case letters

Function Keywords

Certain functions are provided with the ALGOL compiler. Names of these functions can be redefined by the programmer provided no ambiguity results from an attempt to use the identifier both as an ALGOL function and as a programmer variable.

The function keywords are :

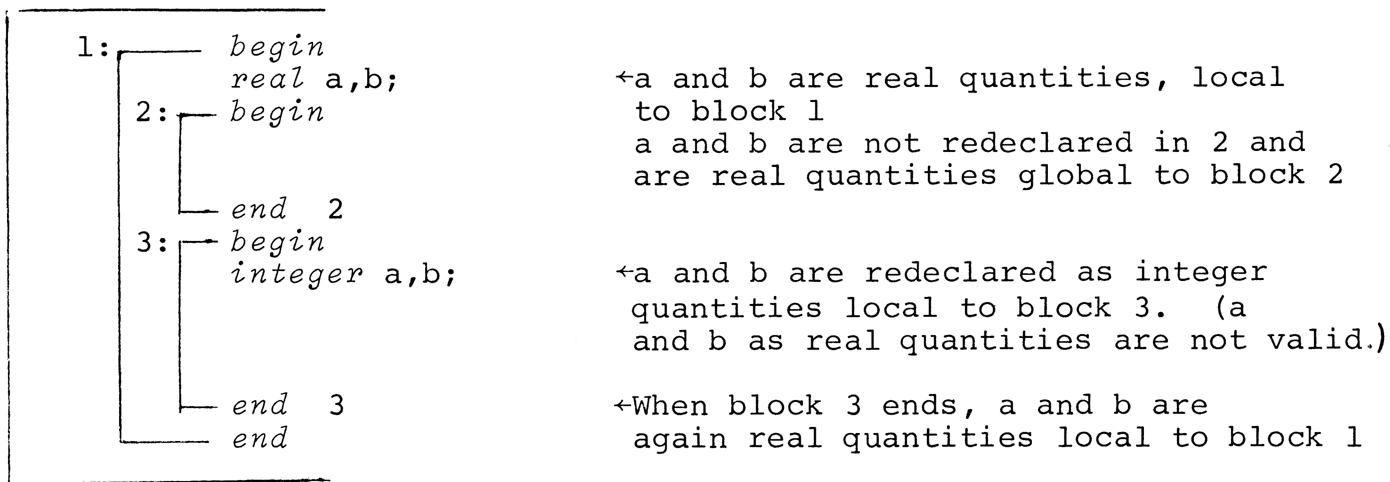
abs	cos	index	rotate	sin
address	entier	length	shift	sqrt
arctan	exp	ln	sign	tab
		page		

SCOPE OF IDENTIFIERS

Simple variables, arrays, labels, switches, and procedures are quantities which have a given scope. Scope is defined as the set of statements and expressions in which the declaration of the identifier associated with the quantity is valid.

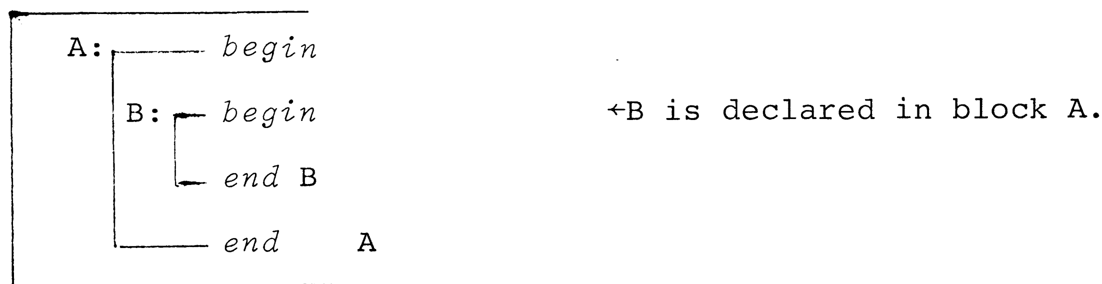
Variables, Arrays, Switches, and Procedures

Variables, arrays, switches, and procedures must be declared, and their scope is the block in which they are declared. By extension, their scope includes inner blocks. An identifier is considered local to the block in which it is declared and global to any inner blocks, unless the identifier is re-declared in an inner block to represent a different quantity as shown in the example.



Labels

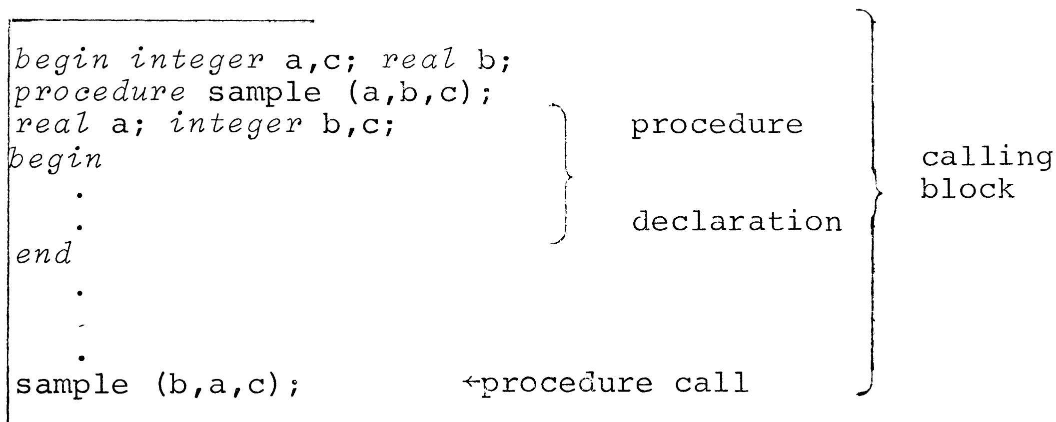
Labels are not defined by a declaration; a label is declared by its use as a label. It is important to note that the label of a block is declared in the block immediately outside the one it serves to label. A label is thus global to the block it labels.



SCOPE OF IDENTIFIERS (continued)

Parameters

Formal parameters that are replaced by name follow the scope conventions of variables. Note that no conflict arises when a formal parameter list is replaced by an actual parameter list containing one or more of the same identifiers but associated with different quantities. The actual parameters simply replace the formal parameters, which have no scope in the real sense of the term.
For example:



Actual parameters a and c replace formal parameters b and c; actual parameter b replaces formal parameter a.

An actual parameter that replaces a formal parameter by value is not altered in the calling procedure because of the call. The called procedure uses a copy of the parameter during procedure execution.

Storage and Scope

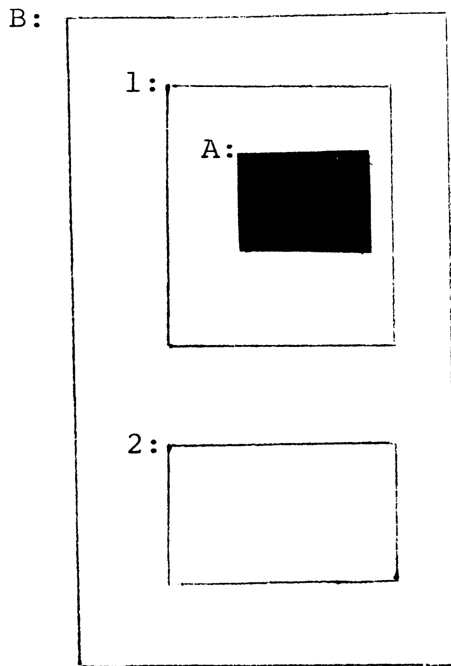
Storage for identifiers is allocated when the identifier is declared and freed when control passes from the block in which the identifier is declared. If an identifier is declared with the *own* declarator, however, storage is allocated when the block is first entered and is not released thereafter.

SCOPE OF IDENTIFIERS (continued)

Scope and Blocks

In the diagram below, presume that each rectangle represents a block and that the labels of the blocks are B, 1,A, and 2. Identifiers declared in block B are defined in all blocks unless a given identifier is redeclared in another block. Identifiers declared in block 1 are defined for blocks 1 and A unless redeclared in A.

Identifiers declared in block A are undefined in any other block; the same is true of the identifiers of block 2. Note that the labels of the blocks are clearly defined in the block outside the block for which they act as labels.



Identifiers declared in darker shaded blocks are undefined in lighter shaded blocks

BLOCKS

In structure, a block is a set of declarations and statements that starts with the keyword *begin* and terminates with the keyword *end*. Semantically, a single block is the smallest set of statements within which a given declaration of an identifier of a quantity is valid for that quantity.

Procedures are also blocks. Procedures usually contain one or more identifier declarations, and every procedure is treated as a block whether or not it contains the keywords *begin* and *end*.

Storage is allocated and deallocated to identifiers dynamically. When a block is entered, storage is allocated to those identifiers declared in the block. Storage for those identifiers is released when exit is made from the block.

Storage is not deallocated when a block inside a block is entered. For such a block, identifiers declared in the outside block remain valid, global quantities. However, the identifiers may be redeclared in an inner block to represent different quantities. If so, the block cannot reference the same identifier outside the block. For example:

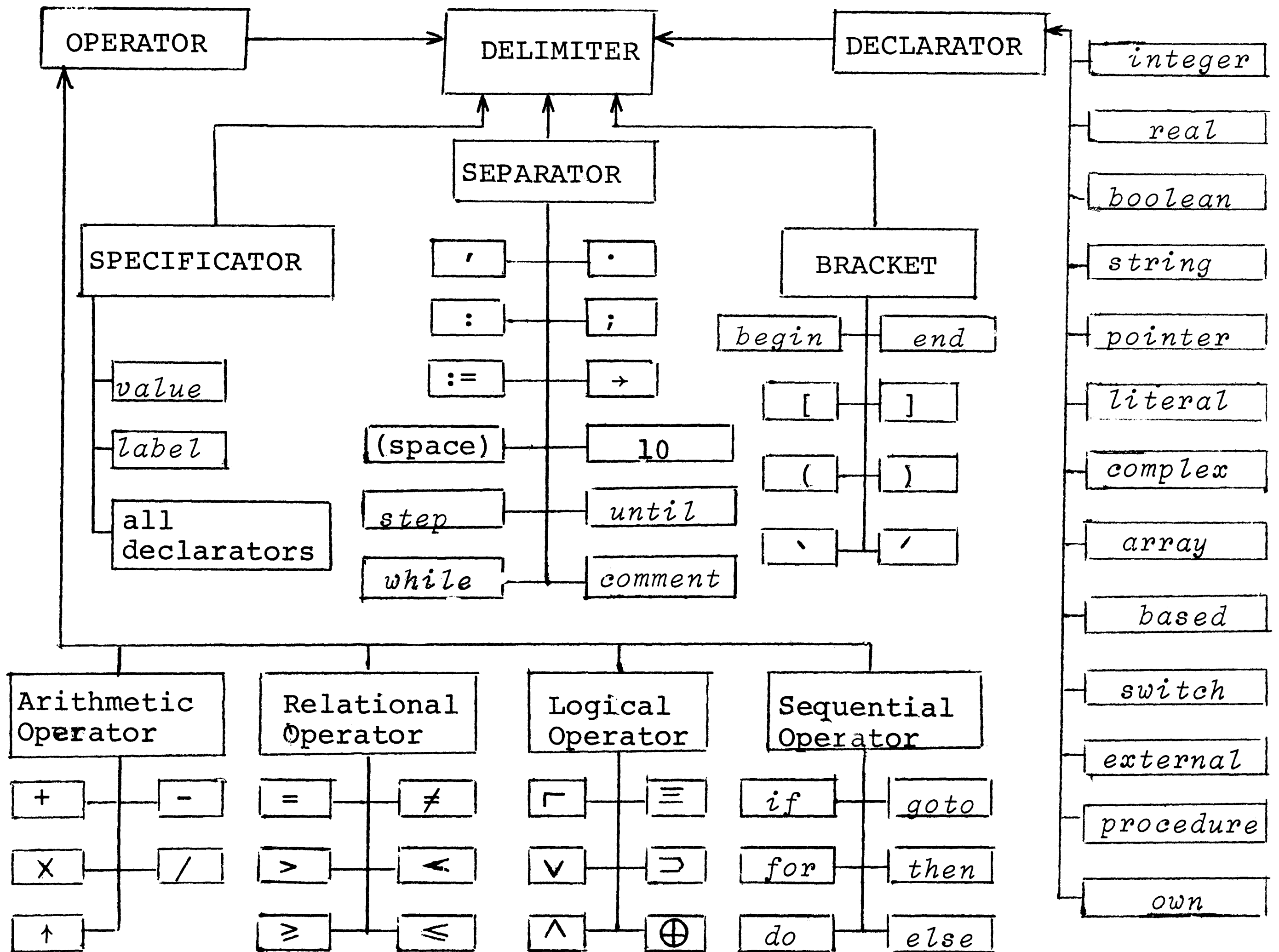
```
A:--      begin
   real x; integer i,j;         ←x is declared as a real quantity
             .
             .                   and Tag is declared a label in block A.
             .
   Tag:  x :=x+sin(x);
             .
             .
             .
   B: ----begin
          real array Tag[i,j];  ←Tag is redeclared an array in block B.
              real z;            z is declared a real quantity in B.
              .
              .
              .
      ------ end B              ←When block B terminates, Tag is again
                                 valid as a label.
              .
              .
              .
     go to Tag;
              .
              .
              .
 --end A
```

BLOCKS (continued)

In the example, x is valid and can be referenced in block A and in block B; z is valid and can be referenced only in block B; and Tag is valid only in A as a label and is valid only in B as an array. Note that the variable dimensions of Tag are valid as integer quantities in both blocks. It is good practice to declare fully every quantity used in a block.

DELIMITERS

The ALGOL delimiters are operators, separators, specificators, declarators, and brackets as shown.



Transliteration of Delimiters

Certain ALGOL delimiters are not found on some teletypewriters. Equivalent symbols are:

ALGOL	TTY	ALGOL	TTY	ALGOL	TTY
∧	*and*	×	*	≠	∧= or <>
∨	*or*	$_{10}$	E or e	→	->
⊃	*imp*	≥	>=or=>	` ´	" "
≡	*eqv*	≤	<=or=<		
¬	*not*				
⊕	*xor*				

Separators

SYMBOL	PURPOSE	EXAMPLE
,	Separate items of lists.	*procedure* rt(a,b,c); *real array* A[i,j,k,l];
.	Decimal point in real numeric values	0.011 25.2
10	Separate base of a number from its power, indicating a power of 10.	$25.2_{10}-3$ $0.01_{10}+6$
:	Terminate a label definition. Separate upper and lower bounds of an array dimension.	a:b: I :=I+2; *real array* A[1:10,-7:0];
;	Terminate a statement, declaration or comment.	*integer array* D[1:20]; *go to* 101; *comment* Transfer to test results.;
:=	Separate variables from the expression to be assigned the variable or variables.	c :=c+1; a :=b :=a-pi/2;
→	Separate a pointer from the based variable to which it points.	ptr→ a
(space)	Separate identifiers and keyword identifiers not otherwise separated by operators and other separators.	*if* a=3 *then go to* 20 *else* a :=b;
step	used in a *for* statement to separate an initial value from an incremental value.	i :=1 *step* 2 . . .
until	used in a *for* statement to separate an incremental value from a terminal value.	i := 1 *step* 2 *until* n...
while	used in a *for* statement to separate a value from a conditional expression.	i :=x+2 *while* a>0 . . .
comment	begin a comment.	*procedure* TCON(a,b,c); *comment*: Test for congruence;

Brackets

SYMBOL	PURPOSE	EXAMPLE
( )	Parentheses enclose formal and actual parameters; enclose the precision of arrays and length of character strings, and enclose expressions to be evaluated.	*procedure* main (a,b,c); *string* (8) B; *integer* (12) *array* B[i,j]; ((A+B)/C)↑2.5
[ ]	Square brackets enclose the dimensions of arrays, subscripts of array elements, and subscripts of switches.	*integer array* M[i,j]; c :=A[1,2]; *go to* B[i];
begin *end*	Keywords *begin* and *end* enclose blocks and compound statements.	*begin real array* act[0:20]; . . . *begin* act[m] :=j; k :=i *end* . . . *end*
\` ´	Grave and acute accents enclose string values. Note that strings may be nested.	\`This is a \`string´
" "	Double quotation marks may also enclose a string value. Use of a single accent mark is possible in a double quote string.	"DON´T GO!"

Arithmetic Operators and Numbers

ARITHMETIC OPERATIONS

Operator	Operation	Resulting Value Type
+	Addition	If both operands are integer the result is integer. If one operand is complex the result is complex. Otherwise, the result is real.
-	Subtraction	
×	Multiplication	
/	Division	
↑	Exponentiation	Permitted combinations and results are described in the table below for real and integer values. Complex base values may be raised to integer values only. Results are defined for complex exponentiation only in those cases in which a real value may be raised to an integer with a defined result.

Base	Exponent	Result
integer real	integer>0	Same as base.
integer real	integer=0	1 1.0
integer=0 real=0	integer≤0	undefined
integer real	integer≤0	real
real>0	real>0	real
real=0	real>0	0.0
real=0	real=0	undefined
real<0	real	undefined

NUMBERS

Numbers are real, integer, or complex. Integers are signed or un-Real numbers may be signed or unsigned, have an optional decimal point and have an optional exponent part.

Integers	Examples of Real Numbers		
0	0	-200.84	$-9.3_{10}{+02}$
1776	1776	0.01	
-25	-25	+.083	$25_{10}{-4}$
			-10^{7}

Numbers having an integral power of 10 can be represented on the teletypewriter with either an upper case E or lower case e replacing the lowered 10.

Real Number	TTY Transliteration
$-976.33_{10}{+2}$	-976.33E+2
$25_{10}{-4}$	25e-4
-10^{7}	-1e7
$_{10}{+02}$	1E+02

Note that in the last two examples, a 1 must appear before E or e to prevent interpretation of this number as an identifier.

To approximate the number of decimal digits of precision that can be stored in a given number of 16-bit words, use the following formulas. n represents the declared precision in words.

integer digits = 5(n-1)+4 $\qquad$ integer range = $\pm 2^{16n-1}$

real digits = 5(n-1)+2 $\qquad$ $10^{-75} \leq$ real range $\leq 10^{78}$

Complex numbers have a real and an imaginary part and are represented as shown below:

-3.5+67.6I $\qquad$ 2.4E8-2.3I $\qquad$ 3.0E-7+7.9E+6I (3.0E-7 = real, +7.9E+6I = imaginary)

Numbers (continued)

A numeric literal with any radix up to and including 10 can be written. The literal is written, followed by the letter R followed by a number giving the radix, for example:

1001R2	Base 2.
.12122R3	Base 3.
77E-6R8	The exponent represents 8^{-6}
.3E+5R4	The exponent represents 4^{+5}

Boolean Operators and Values

RELATIONAL OPERATORS

Symbol	Operation	TTY Symbol
<	less than	same
≤	less than or equal	<= or =<
=	equal	same
>	greater than	same
≥	greater than or equal	>= or =>
≠	not equal	<> or ^=

LOGICAL OPERATORS

Symbol	Operation	TTY Symbol
¬	logical negation	*not*
∧	logical and	*and*
∨	inclusive or	*or*
≡	equivalence	*eqv*
⊃	implication	*imp*
⊕	exclusive or	*xor*

LOGICAL OPERATOR TRUTH TABLE

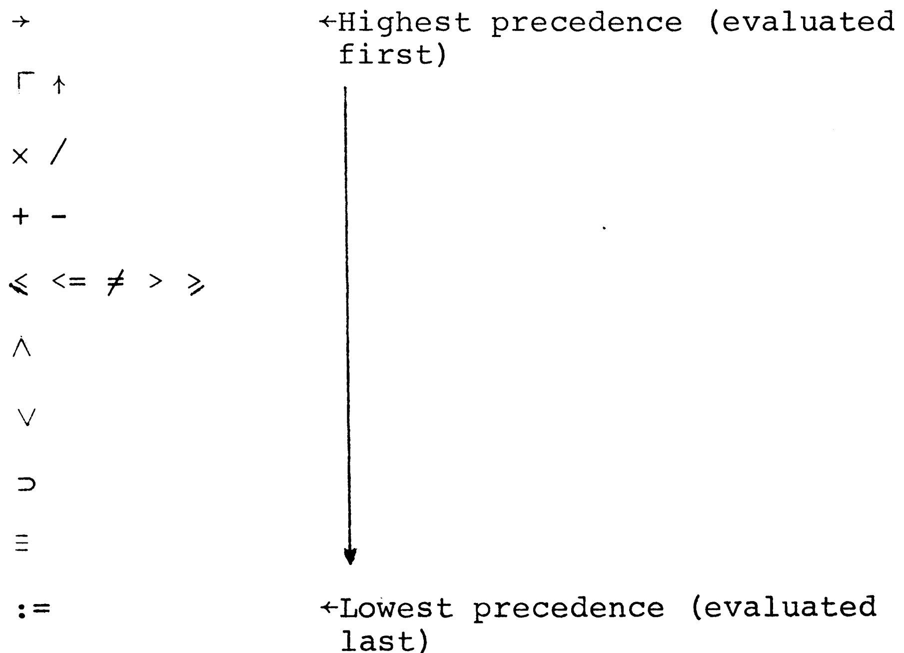
Operands		Operations					
Y	Z	*not* Y	Y *and* Z	Y *or* Z	Y *imp* Z	Y *eqv* Z	Y *xor* Z
false	*false*	*true*	*false*	*false*	*true*	*true*	*false*
false	*true*	*true*	*false*	*true*	*true*	*false*	*true*
true	*false*	*false*	*false*	*true*	*false*	*false*	*true*
true	*true*	*false*	*true*	*true*	*true*	*true*	*false*

Rules of Arithmetic and Boolean Expression Evaluation

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

1. Precedence of operator evaluation

OPERATOR



2. ⌐ and ↑ operations are evaluated from right to left.

3. Parentheses are used to alter the order of operator precedence.
   A parenthesized expression is evaluated as an entity before further evaluation proceeds.

Bit Operations

Bit operations use binary and octal literals combined with logical operators to manipulate bits of integer data.

Operator	Description	Example
A∧B (*and*)	Result is 1 if and only if A is 1 and B is 1 in that bit position.	A :=11001R2; B :=10100R2; A∧B :=10000R2;
⌐A (*not*)	Result is the bit complement of A.	A :=110011R2; ⌐A :=001100R2;
A∨B (*or*)	Result is 1 if either A or B is 1 in that bit position.	A :=100111R2; B :=110000R2; A∨B :=110111R2;
A⊕B (*xor*)	Result is 1 if and only if A and B are complements in that bit position.	A :=100100R2; B :=001101R2; A⊕B :=101001R2;
A≡B (*eqv*)	Result is 1 if and only if A and B have identical bits in that bit position.	A :=100100R2; B :=001101R2; A≡B :=010110R2;
A⊃B (*imp*)	Result is 1 if A is 0 in that bit position or if both A and B are 1 in that bit position.	A :=100100R2; B :=110001R2; A⊃B :=111011R2;

For example, assume x is some integer.

x :=x *and* 111111R2;	First 10 bits of x set to zeroes,
x :=x *and not* 777R8;	Last 7 bits of x set to zeroes,
x :=x *and not* 52525R8;	Alternate bits, beginning at bit 1, are set to zeroes,
x :=x *and* 52525R8;	Alternate bits, beginning at bit 0, are set to zeroes.

Bit Operations (continued)

The coding generated by a bit operation is shown below. Note that the operands are *integer* not Boolean type.

```
; BEGIN INTEGER A,B,C;
; A := B*2 AND 11101R2 OR C;

          LDA     0,S+6,3          ;B
          MOVZL   0,0
          LDA     3,.LP
          LDA     1,5,3            ;LITERAL
          AND     1,0
          COM     0,0
          AND     0,2
          ADC     0,2
          LDA     3,.SP
          LDA     0,S+15,3         ;TEMPORARY
          STA     0,S+5,3          ;A

; END
; END

          JSR     @RETURN
FS1=      16                    ←Frame size.  See Appendixes A
                                 and B.
SP=       100000                ←Stack pointer.
LP:       000012                ←Literal table.
          177773
          000005
          000001
          000002
          000035

          .END
```

EXPRESSIONS

The primary constituents of an ALGOL program - which represents algorithmic processes - are expressions. Expressions are arithmetic, Boolean, designational, or pointer.

Each type of expression may be either a simple expression or a conditional expression. Simple expressions are similar to expressions in other programming languages; conditional expressions are a unique ALGOL feature. In a conditional expression, one out of several expressions (arithmetic, Boolean, designational, or pointer) is selected for evaluation on the basis of the truth value of a Boolean expression in an *if* clause. An *if* clause has the form

if Boolean - expression *then*...

Constituents of expressions (except for certain delimiters such as ( ) and [ ] and :=) are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, sequential and pointer operators. Expressions may be nested to any depth.

Arithmetic Expressions

An arithmetic expression is a rule for computing a numerical value.

A simple arithmetic expression is a collection of one or more numbers, arithmetic variables and function designators combined with arithmetic operators to form a meaningful mathematical expression which always defines a single numerical value. Each variable of the expression must already have a defined value.

Examples:

```
A+B/f        x↑(k-4)×(y-z)          sum↑cos (y+z×3)/7.394  -8
                                                         10
c-d×g↑i      (-b+sqrt (d))/2/a
```

Real numbers are stored in floating-point and integers are stored in fixed point. An arithmetic expression consisting of a real value and an integer value will require conversion of the integer to floating-point. For example:

```
begin real x;
      y :=x+1;        ←conversion required
         .
         .
         .
      y :=x+1.;       ←no conversion required
```

EXPRESSIONS (continued)

Some examples of coding generated by arithmetic expressions are shown below.

```
SUM:=TERM:=DENOM1:=1;

        SUBZL   0,0             ;(1)
        FETR
        FXFL    0,0
        FPRC    2
        FLD3    .SP
        FSTA    0,S+11,3        ;DENOM1
        FSTA    0,S+7,3         ;TERM
        FSTA    0,S+5,3         ;SUM

DENOM2:=N+1;

        FEXT
        LDA     1,S+4,3         ;N
        INC     1,1
        FETR
        FXFL    1,1
        FSTA    1,S+13,3        ;DENOM2

A: TERM:=-TERM*(X/2)^2/(DENOM1*DENOM2);

        FEXT
L1:     FETR
        FLD3    .SP
        FLDA    0,S+7,3         ;TERM
        FNEG    0,0
        FEXT
        LDA     3,.LP
        LDA     0,2,3           ;LITERAL
        FETR
        FXFL    0,1
        FLD3    .SP
        FLDA    2,S+2,3         ;X
        FDIV    2,1
        FMUL    1,1
        FMUL    1,0
        FLDA    1,S+11,3        ;DENOM1
        FLDA    2,S+13,3        ;DENOM2
        FMUL    1,2
        FDIV    2,0
        FSTA    0,S+7,3         ;TERM

SUM:=SUM+TERM;

        FLDA    2,S+5,3         ;SUM
        FADD    2,0
        FSTA    0,S+5,3         ;SUM
```

EXPRESSIONS (continued)

A conditional arithmetic expression contains at least one *if* clause with a Boolean expression, two or more arithmetic expressions, and may contain other sequential operators besides *if* and *then*.

Examples:

```
if g>0 then S+3×Q/A else 2×S+3×q

if a<0 then U+V else if a×b>17 then U/V else if k≠y then V/U else 0

A[i] :=if i<j then B[j]+i
else B[j+1];
```

The subscripts of an array element may be given as simple or conditional arithmetic expressions whose value is an integer. The length of a string or the dimensions of an array can be declared as simple or conditional arithmetic expressions evaluating to integers if the values of variables of the expression are defined when the block is entered.

Examples

```
A [n] :=A [if y<0 then n else n+5];

real array A [i,j,k];
```

EXPRESSIONS (continued)

Boolean Expressions

A Boolean expression is a rule for computing a logical value (*true* or *false*).

Simple Boolean expressions are collections of logical values, Boolean variables and functions, and logical and relational operations. Relational operations consist of simple arithmetic expressions and relational operators.

Example: Assume that A :=*true*; B :=*true*; W :=2; X :=4; Y :=6;

STATEMENT		LOGIC VALUE
D :=*not* A;		*false*
E :=W>X;		*false*
F :=W<X *and* W<Y;	(*true* and *true*)	*true*
G :=W≠X *and not* A;	(*true* and *false*)	*false*
H :=*not* A *or* W=X;	(*false* or *false*)	*false*
J :=*not* (A *and* W>X);	(*not* ((*true* and *false*) i.e., *false*))	*true*

A conditional Boolean expression contains at least one *if* clause and two or more Boolean expressions, and may contain certain other sequential operators besides *if* and *then*.

Examples:

if k<l *then* s>w *else* h<c

if if if a *then* b *else* c *then* d *else* f *then* g *else* h<k

EXPRESSIONS (continued)

Pointer Expressions

A pointer expression is a rule for obtaining a pointer to an address.

A simple pointer expression is a pointer identifier or a subscripted pointer identifier, which may be combined with integer numbers or arithmetic expressions that evaluate to an integer using the arithmetic operators + and -.

A conditional pointer expression contains at least one *if* clause, two or more pointer expressions and may contain other sequential operators besides *if* and *then*.

A pointer expression is often followed by the pointer operator → and a based variable to which the expression points.

Examples:

```
P→ a

(p + 2)→n

if k<l then (p + i)→a else (p + l)→a

p [i]→a
```

Designational Expressions

A designational expression is a rule for obtaining the label of a statement.

A simple designational expression is a label identifier, an unsigned integer used as a label, a subscripted label identifier or a subscripted switch designator. The subscript of a label identifier or switch designator evaluates to an integer value.

A conditional designational expression contains at least one *if* clause, two or more designational expressions and may contain other sequential operators besides *then* and *if*.

Examples:

```
17

p9

Choose [n -1]

TOWN [if y<0 then N else N+1]

if Ab<c then 17 else q [if w≤0 then 2 else n]

go to if a∨b then tag;
```

STATEMENTS

The statement is the basic operating unit of ALGOL. There are six kinds of statements:

NAME	EXAMPLE
assignment	i :=i+1;
conditional (*if*)	*if* i=0 *then* *go to* 25;
transfer (*go to*)	*go to* labelxx;
loop (*for*)	*for* i :=1 *step* 1 *until* n *do*...
procedure call	somefunction (x);
dummy	tag:;

Statements are executed consecutively unless the sequence is broken by an unconditional transfer (*go to* statement) or by some condition that causes a statement sequence to be skipped (*if* statement). Statements may have one or more labels.

Basic statements are often combined to form more complex units of operation, for example, the following combination of assignment, condition, transfer and looping statements:

```
if i >0 then for i :=1 step 1 until n do A[i] :=B[i]+i else go to 25;
```

Each statement within the combination of statements may be labeled:

```
    T1:if i>0 then T2: for i :=1 step 1 until n do
    T3: A[i] :=B[i]+i else T4: go to 25;
```

A further level of freedom in statement sequencing is available. A group of statements can be delimited by *begin* and *end* keywords forming a compound statement. A compound statement is a block in which there are no declarations.

STATEMENTS (continued)

```
Z:  begin integer i,k; real w;
    for i :=1 step 1 until m do
    for k :=i+1 step 1 until m do
        begin w :=A[i,k]; A[i,k] :=A[k,i];   } Compound        } Block
              A[k,i] :=w                      } Statement       }
                                              }                 }
        end                                                     }
        end Z                                                   }
```

Note that a compound statement can contain other compound statements.

Conditional expressions, which may be used wherever a simple expression can be used, provide another degree of freedom. Such constructions as:

go to if...

and *if if...then...else...then...*

are permitted in ALGOL.

Assignment Statement

Format:

```
v :=e;
```

where: v is a variable or list of variables.

e is an expression.

Purpose: To assign the value of the expression on the righthand side of the statement to the variable or list of variables on the lefthand side.

Notes:

1. v may be a subscripted variable.

2. v may be a procedure identifier if the assignment statement appears in the body of the function that defines the procedure identifier.

3. A list of variables on the lefthand side has the format:

```
v1 :=v2 :=...vn :=e;
```

Variables in the list need not have the same data type. The expression is converted to match the data type of each variable, starting at the rightmost. Conversion is made according to the rules given below.

4. The data type of the expression e must be identical to the data type of the variable v except for the following possible conversions:

```
integer v := boolean e;
```

The *boolean* expression is evaluated to 0 or 1. A full word of either 0's of 1's is assigned to v.

```
integer v := real e;
```

The *real* expression is evaluated. The value assigned the variable is entier (e+0.5). See built-in function entier.

Assignment Statement (continued)

integer v := *pointer* e;

A pointer expression evaluates an integer that is one word long and points to some location. The pointer value can be assigned to an integer variable if the variable has the default precision of one word.

boolean v := *integer* e;

The *integer* expression is evaluated. If the expression has a value of 0, the value *false* is assigned to the variable; otherwise, the variable is assigned the value *true*.

pointer v := *integer* e;

The value of the *integer* expression is assigned to the *pointer* variable. The integer must be of default precision (one word).

real v := *integer* e;

The *integer* expression is evaluated and the decimal point is placed after the last digit when assigning a *real* value to v.

string v := *integer* e;

The *integer* expression is evaluated and assigned to *string* v as a string of characters, all of which are digits.

Examples:

```
S[a,k+2] :=3-arctan (S×zeta);
```

Assignment Statement (continued)

The lefthand subscript is first evaluated; the arithmetic expression is evaluated and assigned to S[a,k+2].

```
Boo :=b>c and d;
```

A truth value is assigned to Boo when the Boolean expression b>c *and* d is evaluated.

```
p :=address (f);
```

The pointer p is assigned the address of f.

```
Formula :=diff/(x - 2);
```

Formula is a function procedure and the assignment statement appears in the body of the function.

An example of coding generated by a *real* arithmetic assignment statement is:

```
BEGIN
D:=(1+(T1*OMEGA)^2)*(1+(T2*OMEGA)^2);

   FETR
   FLD3      .SP
   FLDA      0,S+2,3          ;T1
   FLDA      1,S+16,3         ;OMEGA
   FMUL      0,1
   FMUL      1,1
   FEXT
   SUBZL     0,0              ;(1)
   FETR
   FXFL      0,2
   FADD      1,2
   FLDA      1,S+4,3          ;T2
   FLDA      0,S+16,3         ;OMEGA
   FMUL      1,0
   FMUL      0,0
   FETR
   FXFL      0,1
   FADD      0,1
   FMUL      1,2
   FSTA      2,S+20,3         ;D
```

<u>Assignment Statement</u> (continued)

An example of coding generated by a boolean assignment statement is:

```
ANGLES:=ABS(ABC-90)<ANGTOLER AND ABS(BCD-90)

        LDA     3,.LP
        LDA     0,0,3           ;LITERAL
        FETR
        FXFL    0,0
        FPRC    2
        FLDS    .SP
        FLDA    1,S+10,3        ;ABC
        FNEG    0,0
        FADD    1,0
        FABS    0,0
        FLDA    2,S+20,3        ;ANGTOLER
        FSUB#   2,0
        FEXT
        SUBCL   1,1
        FETR
        FXFL    0,0
        FLDA    1,S+12,3        ;BCD
        FNEG    0,0
        FADD    1,0
        FABS    0,0
<ANGTOLER AND ABS (ADC-90)<ANGTOLER AND ABS(BAD-90)<ANGTOLER;

        FSUB#   2,0
        FEXT
        SUBCL   2,2
        AND     2,1
        FETR
        FXFL    0,0
        FLDA    1,S+14,3        ;ADC
        FNEG    0,0
        FADD    1,0
        FABS    0,0
        FSUB#   2,0
        FEXT
        SUBCL   2,2
        AND     2,1
        FETR
        FXFL    0,0
        FLDA    1,S+16,3        ;BAD
        FNEG    0,0
        FADD    1,0
        FABS    0,0
        FSUB#   2,0
        FEXT
        SUBCL   2,2
        AND     2,1
        STA     1,S+24,3        ;ANGLES
```

for Statement

Format:

```
for cv :=list  do s;
```

where: cv is a controlled variable, which may be subscripted.

list is a list of values the controlled variable can assume.

s is a simple or compound statement.

Purpose: To permit repetitive execution of statement s with the controlled variable set to values specified by list.

Notes: 1. list may be a simple list of values or expressions to be evaluated. In addition, list can include *for* clauses. A *for* clause contains either keywords *step* and *until*, or the keyword *while*.

```
for i :=1    step  1         until      10 do A[i] :=i↑i;

         ↑        ↑                     ↑
      initial  increment              final
      value                           value limit
```

The example above is equivalent to the simple list:

```
for i :=1,2,3,4,5,6,7,8,9,10 do A[i] :=i↑i;
```

Values of the list are assigned to i beginning with the leftmost value and terminating with the rightmost value. When the list is exhausted, the next statement in logical sequence will be executed.

A *while* construction is shown in the statement:

```
for j :=0, 1, v×2 while v<n do m:=j/5;
```

Note that the *while* construction is included as part of a simple list. A list may include any of the clause constructs. For example:

```
for j :=i+k,2,i+2,1 step 1 until n, x while x≠0 do...
```

for Statement (continued)

2. The statement following *do* may be a *for* statement or a compound statement that includes a *for* statement, i.e., *for* statements may be nested.

3. Parts of a *for* statement may be labeled, but an attempt to transfer to a label within a *for* statement from outside the statement will cause an undefined result.

Examples:

```
for I :=1 step 2 until n do
    X[I] := X[I]↑2+I;

for k :=0, n do u[k] :=u[k]/2;

for a [bottom] :=min (a[bottom], a[top]) while top>bottom do
    begin top :=top-1;
            bottom :=bottom+1;
            end:
```

for Statement (continued)

An example of coding generated by a *for* statement is:

```
; FOR M:=1 STEP 1 UNTIL N DO NFACT:=M*NFACT;

          FEXT
G4:       SUBZL    0,0              ;(1)
          FETR
          FXFL     0,0
          FLD3     .SP
          FSTA     0,S+17,3         ;M
          FEXT
G5:       LDA      3,.SP
          LDA      0,S+4,3          ;N
          FETR
          FXFL     0,0
          FEXT
          SUBZL    1,1              ;(1)
          FETR
          FXFL     1,1
          FLDA     2,S+17,3         ;M
          FSUB     2,0
          FEXT
          MOV      0,0,SZC
          JMP      G6
          FETR
          FADD     1,2
          FSTA     2,S+17,3         ;M
          FEXT
          JSR      @.+1
          G3
          JSR      @JUMP
          G5
G6:       JSR      @JUMP
          G2
G3:       MOV      3,1              ;SAVE
          FETR
          FLD3     .SP
          FLDA     0,S+17,3         ;M
          FLDA     1,S+15,3         ;NFACT
          FMUL     0,1
          FSTA     1,S+15,3         ;NFACT
          FEXT
          JSR      @JUMP1
```

An example of coding generated by a *for* statement using strings is shown on the following page.

for Statement (continued)

```
; EXTERNAL STRING S;
; EXTERNAL PROCEDURE PRINT;
; FOR S := "YOU", "ME", "THEM" DO

G3:     JSR     @MOVSTR
        S                           ;S
        LP+0                        ;LITERAL
        JSR     @.+1
        G2
G4:     JSR     @MOVSTR
        S                           ;S
        LP+2                        ;LITERAL
        JSR     @.+1
        G2
G5:     JSR     @MOVSTR
        S                           ;S
        LP+4                        ;LITERAL
        JSR     @.+1
        G2
        JSR     @JUMP
        G1
G2:     MOV     3, 1                ;SAVE

; PRINT (S);

        LDA     3, .SP
        STA     1, S+0, 3           ;TEMPORARY
        JSR     @CALL
        PRINT
        2
        S                           ;S
        002602                      ;STRING EXTERNAL
        LDA     1, S+0, 3           ;RETURN
        JSR     @JUMP1

; END

G1:     JSR     @RETURN
FS1=    0

SP=     100000
LP:     ST+0*2     ←Byte pointer to string
        001403     ←String specifier giving current and
        ST+2*2      maximum lengths.  See Appendixes A
        001002      and B.
        ST+4*2
        002004

ST:     .TXT    "YOU"
        .TXT    "ME"
        .TXT    "THEM"

        .END
```

go to Statement

Format:

```
go to d;
```

where: d is a label or designational expression.

Purpose: To transfer to the statement having the label d.

Notes:

1. Transfer cannot be made from outside a block into the block. Transfer can only be made to labels defined locally or globally in the block containing the *go to*.

2. Designational expressions may be:

   a. Labels with a variable subscript.
   b. Switches.
   c. Conditional expressions.

3. If the value of a switch or a label subscript expression is undefined, no transfer occurs and the statement following the *go to* is executed. (A switch is undefined if the value is greater than the number of labels declared for the switch. A label subscript expression is undefined if it evaluates to a subscript for which there is no matching label.)

Examples:

```
go to 10;
```

Transfer is made to the statement labeled 10.

```
go to a[i]
```

i is evaluated and transfer is made to the subscripted label, a[1] a[2]..

```
go to if sin(x)<0.25 then 25 else MLABEL;
```

Transfer is made to the statement labeled 25 if the sine of x is less than .25. Otherwise, transfer is made to the statement labeled MLABEL.

go to Statement (continued)

```
switch F :=lab1, x1, lab2, x2;
    .
    .
    .
go to F[j];
```

If j evaluates to 1, transfer is made to the statement labeled lab1; if j evaluates to 2, transfer is made to the statement labeled x1,etc.

if Statement

Format:

```
if be then uc;
if be then uc else c;
if be then uc else...
```

where: be is a Boolean expression

uc is an unconditional clause, which may be a statement, a compound statement or a block,but cannot contain another *if* statement.

c is any clause, which may be a statement, a compound statement or a block.

Purpose: To provide conditional transfer of program control. If Boolean expression be is true, the unconditional *then* clause is executed. If Boolean expression be is false, the next statement or block after the *then* clause is executed. This may be either the next statement or block in the program or an *else* clause.

Notes:

1. Blocks and statements contained in *then* or *else* clauses may be labeled.
2. Since *else* clauses may contain conditional statements, it is possible to set up a series of conditions for transfer of program control. The series terminates when a Boolean expression is true, causing a *then* clause to execute.

if Statement (continued)

Examples:

```
if i=0 then go to END_PROG;
```

```
if j<kt then
begin
k :=factor [j]+i; j :=j+i;

lab_7: i :=i+1; S[i] :=j; go to 5;
     end          else go to 15;
```

```
if g<0∧h<0 then isign:= -1 else
if g>0∧h<0 then isign:= +1 else 0;
```

An example of coding of an *if* statement, containing a *go to* statement is:

```
IF ANGLES AND ADJSIDES THEN BEGIN

      AND     1,0
      MOVR    0,0
      JSR     @JUMPC
      G1

PRINT("SQUARE"); GOTO NEXTCASE END;

      JSR     @CALL
      PRINT
      2
      LP+4                      ;LITERAL
      000202                    ;STRING
      JSR     @JUMP
      L2
```

IDENTIFIER DECLARATION AND MANIPULATION

Programmers must declare the characteristics of identifiers to be used. Keyword declarators and certain bracketed information are used to define identifier characteristics.

The major characteristics that can be declared for identifiers are their shape, storage class, and data type. It is also possible to declare precision for certain identifiers.

There are three possible identifier shapes -- scalar, array, or procedure. Arrays and procedures must be declared with the *array* declarator or the *procedure* declarator. Scalars are declared by default.

There are seven possible identifier data types -- integer, real, complex, boolean, string, pointer, or label. An identifier must be declared with the declarators *real*, *integer*, *complex*, *boolean*, *pointer* or *string*. A *label* declarator is only required for identifiers that are formal parameters.

There are five possible identifier storage classes -- local, own, based, value, and external. Identifiers must be declared with the *own*, *based*, or *external* declarator. Local storage is assigned by default. Value is assigned to constants. A *value* declarator is only used for formal parameters.

Maximum length of strings must be declared, and *integer* or *real* identifiers can be declared with the number of words of precision. By default, integers are stored in one word and real data in two words. A complex datum is stored as two real data.

Appendix A contains a chart showing the relationships of the identifier characteristics.

Data Types

There are five data type declarators, which are mutually exclusive. Data type declarators are applied to simple variables that return a value of the type declared; to arrays whose elements return values of the type declared; and to function procedures that return a value of the type declared.

DATA TYPE DECLARATOR	PURPOSE	EXAMPLE
real	Declare a variable, array or procedure that returns a numeric value that is not an integer and does not have an imaginary part.	*real* n, pi, m; *real array* a, b, c[i,j]; *real procedure* x;
integer	Declare a variable, array, or procedure that returns an integer numeric value.	*integer array* A[i,J]; *integer* i,J;
complex	Declare a variable, array, or procedure that returns a complex value.	*complex* x; *complex array* C[5,5];
boolean	Declare a variable, array, or procedure that returns a truth value of *true* or *false*.	*boolean* zero, nosolution;
string	Declare a simple variable,array, or procedure that returns a character string value.	*string* (20) char; *string procedure* sym (x,y);
pointer	Declare a variable, array, or procedure that returns a pointer as its value.	*pointer array* LOCUS [i]; *pointer* pl, p2, p3;

Each *boolean* or *pointer* identifier is stored in a single word. By default, each *integer* identifier is stored in one word and each *real* identifier is stored in two words. The real and imaginary parts of a *complex* variable are each stored as a *real* variable, i.e., each part requires two storage words.

The programmer can specify a different number of words of storage for *integer* or *real* identifiers.

real (1) x;
integer (2) y;
real (3) z;

Single-word precision integers must have values in the range $\pm 2^{14}$; they are not checked for overflow. Multi-word integers and real numbers are checked for overflow, and an error message is issued.

Arrays

An array is declared with the following characteristics:

DATA TYPE — *real*, *integer*, *boolean*, *string*, *complex*, or *pointer*. If type is omitted, *real* is understood.

SHAPE — *array*

DIMENSIONS — Enclosed in brackets and separated by commas.

Up to 128 dimensions are permitted.

BOUNDS — Upper and lower bounds for each dimension may be specified by arithmetic expressions. The lower bound must be less than the upper bound. Examples:

A[2:n]	lower bound is constant; upper bound is a variable.
B[-2:15]	constant upper and lower bounds.
C[i,j+2]	only upper bounds are specified.
D[25,50]	Lower bound is 0 by default.

If an expression containing a variable is used in array dimensioning, that variable must be global to the block in which the array declaration appears. The outermost block of a program must have constant array dimensions, unless it is a procedure with array formal parameters.

Subscripts are checked against array bounds and an error message occurs if the subscript exceeds the possible bounds.

PRECISION — An integer enclosed in parentheses may optionally follow an *integer*, *real*, or *complex* data type. The integer specifies the number of 16-bit words in which each element of the array is to be stored. The default values are one word for *integer* and two words for *real* data.

Single-word precision integer arrays must have values in the range $\pm 2^{14}$; they are not checked for overflow. Multi-word precision integer arrays and real arrays are checked for overflow.

An integer enclosed in parentheses must follow a *string* data type. The integer specifies the maximum number of characters that each element of the array may contain.

Example:

real	(3)	*array*	a, b, c, d	[i, j, k];
↑	↑	↑	↑	↑
data type	3-word precision of each array element	array	identifiers of arrays	3-dimensional arrays of variable bounds

The dimension declaration applies to the entire preceding list of arrays in the example above.

```
integer array ORG [ -10;10, 0:20];
```

ORG is a 21x21 two-dimensional array. Note that negative values may be used in determining upper and lower array bounds.

Each element of an array is a subscripted variable of the form:

```
array_name [sub1, sub2,...subn]
```

where: array_name is the name of the array
sub1...subn is a list of subscripts identifying the array element.

For example:

```
A[25] B[i,j] C[x+10]
```

could all be array elements.

An array element has the effect of a simple variable. The subscripts are evaluated and if necessary converted to type *integer* by the formula: entier (subscript_value +0.5).

The address of each array element may, if desired, be accessed by pointer manipulation.

The most common use of arrays is in loop manipulation. See *for* statement.

Arrays (continued)

An example of coding generated by bit string manipulation of integer arrays is:

```
; INTEGER ARRAY I[10]; INTEGER R;

        SUBO    0,0
        JSR     @ARRAY
        3
        SP+0                    ;I
        010021                  ;INTEGER LOCAL ARRAY
        SP+10                   ;TEMPORARY
        LP+0                    ;LITERAL

; INTEGER ARRAY J[-5:+5];

        JSR     @ARRAY
        3
        SP+3                    ;J
        010021                  ;INTEGER LOCAL ARRAY
        LP+1                    ;LITERAL
        LP+2                    ;LITERAL

; I[R] := I[R+1] AND NOT I[J[R]];

        JSR     @SUBSCRIPT
        2
        SP+11                   ;TEMPORARY
        SP+0                    ;I
        SP+2                    ;R
        LDA     3,.SP
        LDA     0,S+2,3         ;R
        INC     0,0
        JSR     @SUBSCRIPT
        2
        SP+12                   ;TEMPORARY
        SP+0                    ;I
        SP+13                   ;TEMPORARY
        JSR     @SUBSCRIPT
        2
        SP+14                   ;TEMPORARY
        SP+3                    ;J
        SP+2                    ;R
        JSR     @SUBSCRIPT
        2
        SP+15                   ;TEMPORARY
        SP+0                    ;I
        SP+14                   ;TEMPORARY
        LDA     0,@S+15,3       ;TEMPORARY
        COM     0,0
        LDA     1,@S+12,3       ;TEMPORARY
        AND     0,1
        STA     1,@S+11,3       ;TEMPORARY
```

Character String Variables and Arrays

A character string variable must be declared with the *string* data type. All string variables (except formal parameters) must include the maximum length of the string in the declaration.

Example:

```
string (10) a;
```

where: a is a string variable whose value starts at character 1 and can be up to 10 characters in length.

The first character position of a string must be 1.

The maximum length applies to all string variables in the declaration.

```
string (20) g,h,i;
```

g,h, and i all have a maximum of 20 characters.

The value of a string variable is a character string. In input, string values are delimited by accent marks as shown:

```
`$25.00 FOR EACH.´
```

String values may be nested to any depth.

```
`This `string´ is nested.´
```

Accent marks may be transliterated as quotation marks ("string").

When a string is referenced, one or more of the characters of the value may be manipulated. Example:

```
string (8) x; ←declaration of x
```

Assume the current value of x is ABCDEFGH.

```
x(1,8)    ←references the entire string ABCDEFGH
x(1,3)    ←references ABC
x(5,8)    ←references EFGH
x(4,4)    ←references D
```

Character String Variables and Arrays (continued)

Thus, the first number gives the starting character within the string and the second gives the number of characters desired.

An array of character strings can be declared. Each element of the array must have the same maximum length. For example:

```
string (2) symb[1:100];        ←each element of symb has a maximum
                                of two characters.
```

Each element of a string array can be referenced; however substrings of array elements cannot be referenced. For example:

```
string (5) char;               ←simple string variable
string (5) CHAR[6,10];         ←string array
      .
      .
      .
y :=char (1,3 );               ←[1,3] represents a substring of
                                     characters 1 through 3 of
                                      char.
x :=CHAR[1,3];                 ←[1,3] represents element CHAR[1,3]
                                     of the array CHAR.
```

Two built-in functions are commonly used in string manipulation. These are the length function and the index function. The length function has a string variable as an argument and returns the number of characters in the string as a value. The index function searches a specified string variable (argument 1) for a given character configuration (argument 2) and returns as a value the starting location in the string of the first character of the configuration.

Examples:

```
  string (4) v;
v :='abcd';
i :=length (v);          ←i :=4;
j :=index (v,'cd');      ←j :=3;
```

Character String Variables (continued)

Some examples of how strings may be used are:

Pattern Match and Replacement

```
i :=index (a,` ´ );
j :=index (a(i+1), ` ´ );
comment search string a for some character
        delimited by blanks;
for k :=i+1 step 1 until j-1 do
    a(k,k) := `*´ ;
comment replace the character with an asterisk character;
```

Editor Command Table Lookup

```
string (10) commands;
switch S :=Top, Search, Append, Insert;
commands := `TSAI´ ;
loop: i :=index(commands, (Readchar));
     comment : Readchar is a function that reads
               a character;
            if i=0 then fail else S[i];
      error ( `illegal command´ );
      go to loop;
```

Pointers and the *based* Declarator

Pointers, *based* declarators, and the built-in address function represent three extensions to the ALGOL language that provide close programmer control over manipulation of addresses of variables.

These extensions allow the programmer to indirectly reference a program variable by a pointer.

The programmer must declare at least one *pointer* variable and one *based* variable of the data type of the variable whose address is to be manipulated. Using the address function, the pointer is then set to point to the address of some program variable.

The *based* variable has all the declared characteristics of the program variable but it is not stored in core when storage is allocated for the block. Once the pointer is set to an address, the identifier of the *based* variable can be substituted in statements manipulating the address. The pointer must appear in these statements as shown:

```
  begin real x,z;
  based real y;
     pointer p;
       .
       .
       .
p :=address (x);

       .
       .
       .
p→y :=p→y+2;            ←statement is equivalent to x :=x+2;
```

The *based* variable can be considered a template of the program variable. As long as the pointer is set to x, the pointer and *based* variable can be used to modify the address. In this way the programmer can perform address modification and manipulation at very little cost in code generation.

The pointer can be reset to z, and the *based* variable can then be used in a similar way, representing program variable z.

Pointers and the *based* Declarator (continued)

Example:

```
B:  begin real  x,z; real based y;  ←x,y, and z are all real.  y
  pointer p;                         is declared based.  p is de-
        .                            clared a pointer
        .
        .

   11:  p :=address (x);            ←pointer p is assigned the
        .                            address of x.
        .
        .

   22:  p→y :=p→y+2;                ←based variable y is superimposed
        .                            upon x.  Statement 22 is the
        .                            equivalent of x :=x+2;
        .

   33:  p :=address (z);            ←p is reassigned the address of z.
        .
        .
        .

   44:  p→y :=p→y+3;                ←based variable y is superimposed
                                     upon z.  Statement 44 is equiv-
                                     alent to z :=z+3;
```

The program variable referenced by the pointer can be a simple variable, an array, or an element of an array. For example:

```
begin pointer a; integer array b; integer i;
      based integer x;
        .
        .
        .
      a :=address (b[i]);           ←pointer a is assigned the address
                                     of array element b [i];
        .
        .
        .

      b[i+1] :=a→x;                 ←x is effectively superimposed up-
        .                            on b[i] in the expression, which
        .                            is equivalent to: b[i+1] :=b[i];
        .
       end
```

Pointers and the *based* Declarator (continued)

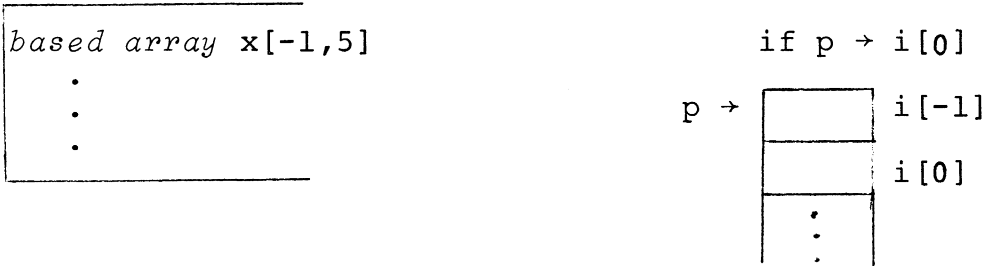
Pointer arrays may be declared and *pointer* expressions may be used in address manipulation. For example:

```
  begin pointer A; real c; based real b;
        .
        .
        .
A :=address (c);
        .
        .
        .
(A+1)→b :=0;           ←location c+l is set to 0
        .
        .
        .
```

It is possible to assign an address to a pointer and then substitute a *based* variable without using the address function.

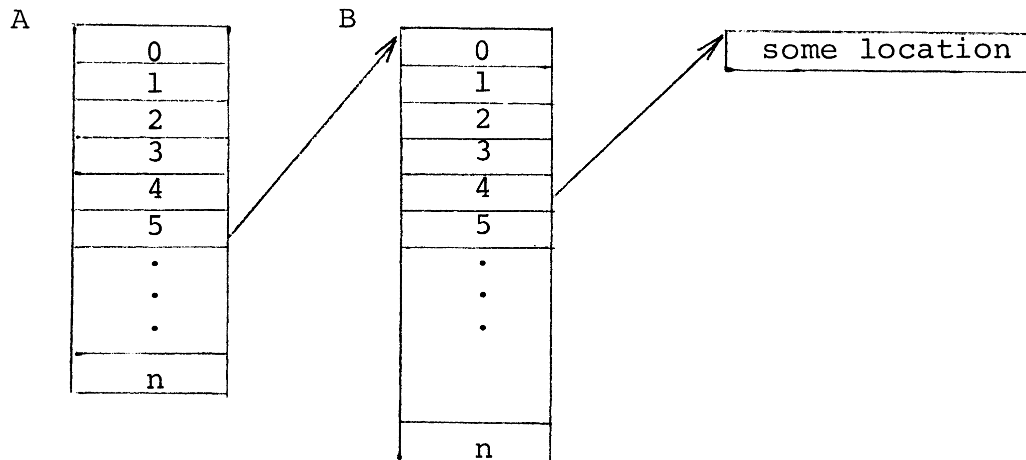
```
begin pointer array A[n];
based pointer array B[n];
based integer i;
        .
        .
        .
p :=A[5]→B[4];         ←pointer array element A[5] points to a
        .                based pointer array element B[4]. The
        .                pointer value assigned to p can later point
        .                to some other based variable, such as i.
y :=p→i;
        .
        .
        .
```

Note: When using a *based* array, it is assumed that the pointer always points to the first word of data in the array, e.g.:



Pointers and the *based* Declarator (continued)

The diagram of the arrays of pointers indicates the assignment of p:



Use of the pointer feature can be shown in list processing. Suppose the programmer wishes to search a singly threaded list, list_x, for a location, key.

```
begin pointer list_x;          ←pointer and integer are interchangeable
based integer i;
      .
      .
      .
p :=address (list_x);
      .
      .
      .
LOOP: go to if((p:=p→i)=0) then EXIT   ←if key does not exist
         else if (p+1)→i =key then EXIT ←if key is found
         else LOOP;
              .
              .
              .
EXIT:...;
```

Pointers and the *based* Declarator (continued)

A threaded list containing backward, and forward pointers can be set up using pointer referencing. Assume in the following example that allocate is a function that allocates a new element consisting of one or more words.

```
p :=allocate;              comment: allocate a new word element;
 p→i  :=list_1;            comment: thread forward pointer (word 0)
                                    of new element;
(p+1)→i :=0;               comment: set backward pointer (word 1)
                                    of new element;
if list_1 ≠ 0 then
begin (list_1+1)→i :=p;    comment: backward pointer;
       list_1 :=p;
end                        comment: list_1 may be 0; if not, pointers
                                    thread the list;
```

Use of pointers may seem awkward in source language but is recommended for programs where efficient code is desired.

Labels

A label is an unsigned integer or an identifier. Blocks and statements, including statements within compound statements, may have labels. The label is delimited from the block or statement by a terminating colon. A block or statement may have more than one label.

```
A:      ┌─ begin
        │    .
        │    .
        │    .
        │  15: a1: x :=x+1;
        │    .
        │    .
        │    .
        │  go to 15;
        │    .
        │    .
        │    .
        └─ end
```

Unlike other identifiers, a label does not require a special declaration; its appearance before a block or statement constitutes a declaration. A label is declared in its smallest enclosing block.

Labels (continued)

```
B:  ┌──── begin real s;
    │         .
    │         .
    │         .
    │    A:┌─ begin real z;      ←A (like s) is declared in Block B
    │      │    .                  and is valid in both blocks A and
    │      │    .                  B.
    │      │    .
    │      │  x: z :=z/s;
    │      │    .                  x(like z) is declared in block A
    │      │    .                  and is valid only in block A.
    │      └─ end A
    └────       end B
```

A dummy statement may be written in ALGOL. A dummy statement provides only a label to which a transfer can be made. For example, a transfer can be made to a labeled *end* delimiter terminating a compound statement or block.

```
begin
  .
  .
  .
if j ≠ 0 then go to Z;
  .
  .
  .
Z: end                          ←labeled end
```

A label may be subscripted with a simple integer subscript. If a block contains ten labels, a[1[, a[2],...a[10],execution of the statement:

```
go to a[j ];
```

causes j to be evaluated and transfer to be made to the corresponding statement label. If j evaluates to a value outside the range of statement labels, e.g., 25, then the next consecutive statement after

Labels (continued)

the *go to* is executed.

Any unsigned integer may be used as a label subscript. However, the amount of storage reserved for subscripted labels will be large enough to contain all subscripts smaller than the largest subscript. For example, a block having two labels named b[1] and b[100] requires 50 times as much storage area for labels as a block having two labels b[1] and b[2].

Switches

Switches are variables that identify a number of alternate labels to which program control may transfer. A switch is declared with a list of labels and designational expressions. The position occupied by a label or designational expression in the list determines whether that label is the one to which transfer is made.

Examples:

```
switch TESTPROG :=a,b,if x>0 then i else d,10,5,c,8,op,3,y3;
```

Switch TESTPROG is defined with 10 alternate labels or expressions evaluating to a label, where a has a position value of 1 and y3 has a position value of 10. If the following statement is encountered during execution:

```
go to TESTPROG [j];
```

j is evaluated. If j=2, transfer is made to label b; if j=3, transfer is made to either label i or d, based upon the evaluation of the designational expression.

```
switch SF :=a,bl,bw,c,d,7;     ←declaration of switch SF
          :
     go to SF [i];             ←transfer to one of the labels
```

In this example i will be evaluated. If i=1, transfer is made to the statement labeled a, if i=2, transfer is made to the statement labeled bl, etc.

If a switch variable evaluates to a value that is outside the range of the switch, the next statement after the *go to* is executed. For

Switches (continued)

example, in the second example, on the previous page, there are 6 possible values for i--1,2,3,4,5, or 6. If i evaluates to a larger integer, the next statement after the *go to* is executed.

own Declarator

Storage for a block is dynamic. Identifiers declared within a block are allocated storage when the block is entered, and storage is released at the time of exit from the block. If a block is entered more than once during execution of a program, variables will be undefined each time the block is entered.

The *own* declarator allows the programmer to specify a variable or variables whose value at the time of exit from the block will be retained. When the block is subsequently reentered, *own* variables are defined.

Example:

```
a: begin integer i, j; own real Hs, s;
        .
        .
        .
   end
```

Each time block a is entered, variables i, and j are undefined. However, after the first execution of a, variables Hs and s have a specified value each time a is entered, the values being that of Hs and s at the time the block was last exited.

external Declarator

Variables may be external to a given program. Such variables must be stored in an external area by assembly. They can be used in a given program if the external variable is declared *external* in the program in which it is used.

Example:

```
al:     begin external integer k;
              integer i, j;
         .
         .
         .
        end al;
```

Literals

Literals are identifiers that are declared with a given value. They provide a means of generating constants with names, so code will be efficient and all occurrences of a constant may be modified in one place. For example:

```
begin
literal MAX(100);              ←100 will replace all occurrences of MAX
literal size(MAX);              in the block. If the parenthesized val-
integer array X [0 :MAX];       ue is changed, all occurrences of MAX
    :                           will be changed.
    :
```

Literals adhere to block structure. A literal declared in an outer block will be local to that block and global to all inner blocks in which the literal declaration is unchanged.

An identifier declared with a literal value in an outer block can be redeclared with another value in an inner block.

```
  begin literal R(0);
        .
        .
        .
  begin literal Z(0), R(1);
```

Any legal value may appear in a literal declaration.

```
  begin literal Y(true), s("A-1023");
```

Literals (continued)

An example of coding and use of literals is:

```
; LITERAL PI(3.14159), FREQ(1), FLAG13(100R2), FLAG14(10R2);
; LITERAL PHASE(PI);
; INTEGER I; REAL (4) X;
; I:=I+FREQ;

        LDA     3,.SP
        LDA     0,S+0,3         ;I
        INC     0,0
        STA     0,S+0,3         ;I      <-literal is "remembered"

; X:=X↑5+I↑2;

        FETR
        FPRC    4
        FLDA    0,S+1,3         ;X
        FALG    0,1
        FEXT
        LDA     3,.LP
        LDA     1,6,3           ;LITERAL
        FETR
        FXFL    1,2
        FMPY    1,2
        FEXP    2,2
        FEXT
        MOV     0,2
        MOV     0,1
        SUBO    0,0
        MPY
        FETR
        FXFL    1,0
        FADD    2,0
        FSTA    0,S+1,3         ;X

; END
```

PROCEDURES

A procedure is a block of code that is executed only when it is called from another block and which returns to the other block when procedure execution is complete. There are two kinds of procedures and procedure calls.

A procedure can be called by a procedure statement in the calling block. The procedure executes and returns to the statement following the procedure statement. Such an ALGOL procedure is similar to subroutines of FORTRAN and other compiler languages.

A procedure can be called by a function reference contained in a statement in the calling block, for example in an assignment statement. Such a procedure returns a value of a given data type to the point at which it was referenced. Such an ALGOL procedure is similar to functions of other compiler languages such as FORTRAN.

Procedure Declarations

The declaration of a procedure consists of defining:

1. The procedure identifier.

2. A procedure data type (if the procedure identifier represents value, i.e., a function procedure.)

3. A list of formal parameters (if actual parameters are to be passed to the procedure when it is called.)

4. Specification of characteristics of the formal parameters.

5. The body of the procedure, which consists of a simple statement, a compound statement, or a block.

Items 1 to 4 constitute the heading of the procedure.

The usual rules of local and global identifiers apply to procedures, i.e., procedures can contain other blocks. An example of a procedure declaration is:

```
real procedure arcsin(x);
         real x;
              arcsin :=arctan (x/sqrt (1-x↑2));
```

Procedure Declarations (continued)

The procedure identifier is arcsin. Arcsin is a function that returns a real value. There is a single formal parameter x, which is specified *real*. The statement body consists of the single assignment statement.

Below is an example of a declaration of a procedure that is not called as a function.

```
procedure innerproduct (a,b,n, sigma);
          comment: compute innerproduct of vectors a and b with
          n components each.  Store result as sigma;
        array a,b; integer n; real sigma;
        begin integer k;
        sigma :=0;                          block within innerproduct
        for k :=1 step 1 until n do
          sigma :=sigma +a[k]×b[k] end
        end innerproduct
```

Note that in the example above the procedure innerproduct contains a begin block. Whether a block is a begin block or a procedure block, the rules for local and global identifiers are the same. In the example, integer k is local to the begin block and is undefined in the outer procedure block. Arrays a and b, integer n, and real variable sigma are global to the begin block.

Many procedure declarations include formal parameters that are replaced by actual parameters when the procedure is called. However, procedures need not have parameters; for example, a procedure that generates a random number would not require that parameters be passed.

A procedure, like a variable, must be declared in the block in which it is used (that is, called). This means that the calling block must include the procedure declaration, including the full text of the procedure body, as part of the declarations at the beginning of the block, except under the conditions noted in the next section.

All ALGOL procedures are recursive and reentrant.

External Procedures

The declaration of a procedure can be compiled as a separate entity. Such a procedure is called an *external* procedure since it is not declared in some other block.

To be called from some other block, the name of the procedure and its *external* characteristic must be declared in the calling block. For example:

CALLING BLOCK

```
begin real x;
integer y;
external real procedure arcsin;
          .
          .
          .
     z :=x × arcsin (x);
```

PROCEDURE

```
real procedure arcsin (x);
          real  x;
arcsin :=arctan (x/sqrt (1-x↑2));
```

Like *external* variables, *external* procedures can be called (used by) a number of blocks to which they are declared to be *external*.

Procedure Calls

Calls to procedures are of two forms: procedure statements and function references.

A procedure statement has the form:

```
procedure_name (p1,p2,...pn);
```

where: procedure_name is the identifier of the procedure

p1,p2,...pn — is a list of actual parameters that replace the formal parameters given in the procedure declaration. The list may be empty.

A procedure statement causes transfer of control to the named procedure and execution of the procedure body using the actual parameters of the calling statement. When the procedure body has been executed, control returns to the calling block at the statement following the procedure statement.

Procedure Calls (continued)

An example of a procedure call is:

```
begin  real  a,b;  real  array  A[1:100];

  procedure  sub_one(a,b,A);                  }
        real    a,b; real array A;            } PROCEDURE       } CALLING
             begin                            } DECLARATION     } BLOCK
               .
               .
               .
             end
               .
               .
               .
             sub_one(a,b,A)        ←PROCEDURE CALL
         X: ---
```

When sub_one has been executed, control returns to the statement labeled X.

A function reference has the form:

```
...procedure_name(p1,p2,...pn)...
```

where:

procedure_name	is the identifier of the procedure.
p1,p2,...pn	is a list of actual parameters that replace the formal parameters given in the procedure declaration. This list may be empty.
...	indicate that the function reference is part of a statement.

A function reference causes transfer of control to the named procedure and execution of the procedure body using actual parameters. When the procedure body has been executed, a value for the procedure is returned to the calling statement. An example of a function reference is:

```
         begin  real  y;
                  .
                  .
real procedure arctan(x); real x;}     PROCEDURE        }
                  .                }   DECLARATION      }
                  .                }                    } CALLING
                  .                                     } BLOCK
                  .
                  .
                  .
     z :=0.215 X arctan (y);          ←PROCEDURE        }
                                        CALL
```

Calling a Procedure by Name and by Value

When an actual parameter is substituted for a formal parameter, the actual parameter may be some variable whose value when passed will be altered one or more times in the course of execution of the called procedure. If so, this is a call by name. The values of certain input variables to the procedures, however, will not be altered in the course of executing the called procedure. When such a parameter is passed, it constitutes a call by value.

Formal parameters that are consistently called by value are given the *value* specificator in the procedure declaration.

Example:

```
real procedure tan (x); value x; real x;
     tan :=sin (x)/cos (x);
```

The actual parameter to be substituted for x in the example is an input value that is unaltered in computing the tangent function.

Sometimes it is desirable to pass a parameter by value to a procedure that does not include a *value* specificator. In that case the actual parameter in the calling procedure is enclosed in double parentheses to indicate a by value assignment. Example:

```
begin  integer  input;
          .
          .
          .
         Routine((input));
          .
          .
          .
```

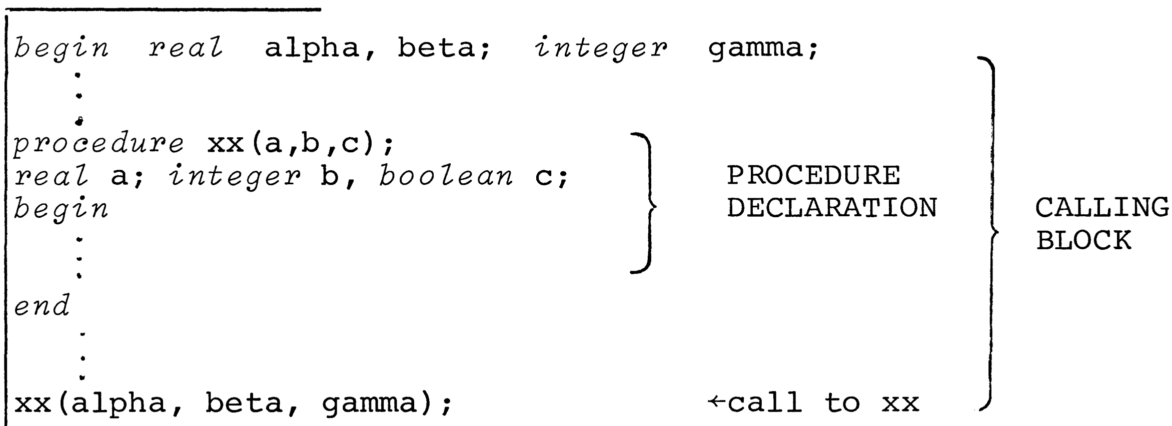
When a function identifier is passed as a parameter, the distinction between by name and by value call is as shown below:

```
begin external integer procedure input;

Routine1  (input);            ←call to Routine1.  The address of
                               input is passed.

Routine2 ((input));           ←call to Routine2.  Function input
                               is called as the parameter of
                               Routine2.
```

Formal and Actual Parameters

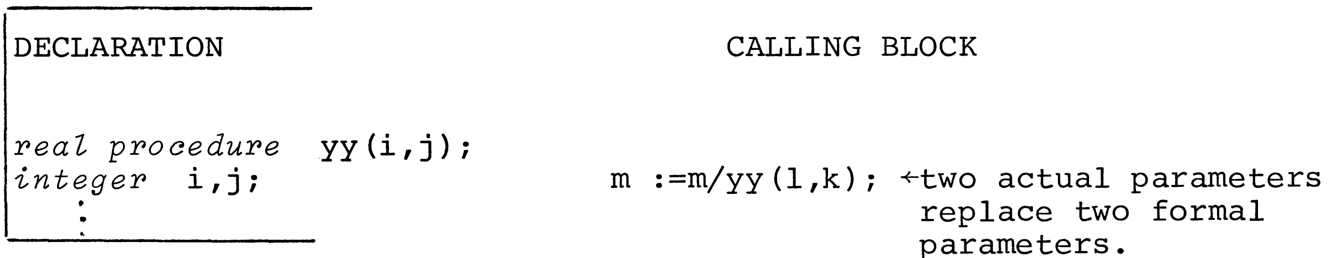
The formal parameters that appear in a procedure declaration are replaced by actual parameters when the procedure is called. Actual parameters may be values or variables, but they must match the formal parameters of the declaration as shown in the following rules:

1. Data types of actual parameters must be compatible with those of formal parameters, i.e., the data types must either match or follow the permitted rules of conversion described on pages 6-3 and 6-4.



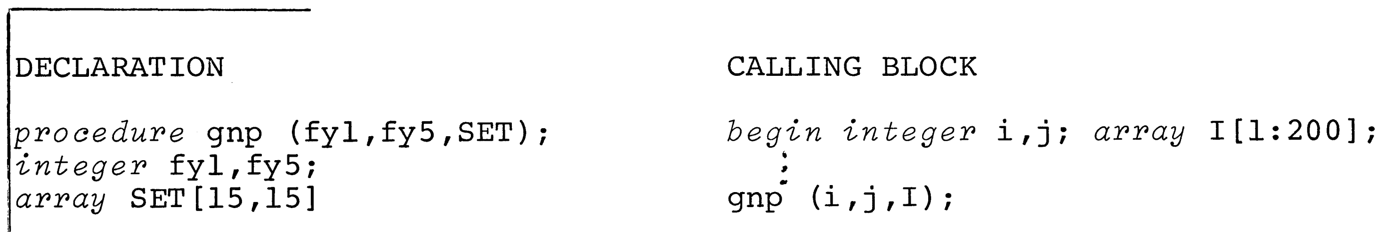
```
begin real alpha, beta; integer gamma;
   .
   .
   .
procedure xx(a,b,c);                    }
real a; integer b, boolean c;           }  PROCEDURE
begin                                   }  DECLARATION      CALLING
   .                                    }                   BLOCK
   .
   .
end
   .
   .
   .
xx(alpha, beta, gamma);                 ←call to xx
```

2. The number of actual parameters in a parameter list must match the number of formal parameters.



```
DECLARATION                        CALLING BLOCK

real procedure  yy(i,j);
integer  i,j;                      m :=m/yy(1,k); ←two actual parameters
   .                                                replace two formal
   .                                                parameters.
```

3. If a formal parameter is an array, it must be replaced by an actual parameter that is an array having the same or fewer array elements.



```
DECLARATION                        CALLING BLOCK

procedure gnp (fy1,fy5,SET);       begin integer i,j; array I[1:200];
integer fy1,fy5;                      :
array SET[15,15]                   gnp (i,j,I);
```

In the example, I[1] replaces SET[0,0], I[2] replaces SET[1,0], ..., I[199] replaces SET[6,10],and I[200] replaces SET[7,10].

Formal and Actual Parameters (continued)

4. A formal parameter that is called by value cannot be a switch identifier or a procedure identifier. An exception is a procedure identifier that has no formal parameters and that defines the value of a function designator. For example, if part of the declaration of procedure x is:

```
procedure x(dd); value dd;      integer dd; begin
```

and if x is called by:

```
x(FD);                    ←where FD is a procedure, then FD must have
                           the form:
```

```
integer procedure FD;     ←no parameters
         :
FD :=...;                 ←FD is assigned some value.
```

5. A formal parameter that occurs on the lefthand side of an assignment statement and is not called by value must be replaced by an actual parameter that is a variable. This rule is a logical extension of the rules of assignment statements.

6. Specification of formal parameters may place further restrictions upon the actual parameters associated with them. Such restrictions must also be observed in the body of the procedure.

7. Any comma in a parameter list can be replaced by the sequence:

)string:(

where: string is any string that does not contain delimiters; or : or keywords *end* or *else*.

The following declarations are equivalent:

```
procedure absmax (a,n,m,y,i,k);
procedure absmax (a) size: (n,m) result: (y) subscripts: (i,k);
```

8. The value of a function is parameter Ø. The following are equivalent, where x is a function:

x(A); or A :=x;

Specificators of Formal Parameters

Characteristics of formal parameters are specified in the procedure declaration as shown in preceding examples of procedure declaration. The parameter rules indicate that there must be a match between data types of formal and actual parameters and a match on the shape, i.e., a simple variable cannot replace a formal parameter that is used as an array.

The keyword declarators are also used as specificators. In addition there are the previously described *value* specificator and the *label* specificator, which allows the programmer to pass a label identifier as an actual parameter.

Procedure Coding Example

An example of coding generated for a procedure is shown below.

```
; REAL (4) PROCEDURE TAN(X); VALUE X; REAL (4) X;

        .EXTU
        .ENT    TAN

        .NREL
TAN:    JSR     @SAVE
        FS1
        2
        SP+Ø                    ;TAN
        ØØ1Ø44                  ;REAL PARAMETER
        SP+4                    ;X
        ØØ2Ø44                  ;REAL VALUE

; TAN:=SIN(X)/COS(X);

        FETR
        FPRC    4
        FLD3    .SP
        FLDA    Ø,S+4,3         ;X
        FSIN    Ø,1
        FCOS    Ø,2
        FDVD    2,1
        FSTA    2,S+Ø,3         ;TAN
        FEXT
        JSR     @RETURN
FS1=    1Ø

SP=     1ØØØØØ

        .END
```

BUILT-IN FUNCTIONS

Certain functions are supplied with the ALGOL compiler.

Mathematical Functions

Arguments to the mathematical functions may be *real*, *complex* or *integer*; each function yields a *real* or *complex* value, with the exception of the sign function which yields an *integer* value.

FORMAT	MEANING AND EXAMPLES	
abs (exp)	Absolute value of expression, exp.	abs (g)↑(i/m)
arctan (exp)	Principal value of $\tan^{-1}$ (exp), where expression exp is in radians.	arctan (y-x)
cos (exp)	Cosine of expression exp, where exp is in radians.	cos (n-pi/2)
exp (exp)	Exponential function of the value of exp, which is the value of the Eulerian constant e raised to exp: e^{exp}	exp (a[10])
ln (exp)	Natural logarithm of expression exp.	ln (a/2)
sign (exp)	Sign of expression exp, which is: +1 for exp>0 0 for exp=0 -1 for exp<0	sign (a/b)
sin (exp)	Sine of expression exp where exp is in radians.	sin (omega × t)
sqrt (exp)	Square root of expression exp.	sqrt (abs (x-y))

BUILT-IN FUNCTIONS (continued)

Transfer Function

The entier function "transfers" a *real* expression to an *integer* expression.

FORMAT	MEANING AND EXAMPLES	
entier (exp)	The largest integer value not greater than the value of real expression exp.	entier (abs(a/b))

Address Function

The address function permits assignment of the location of a variable as the value of a pointer. The function has the form:

address (v)

where: v is some subscripted or unsubscripted program variable.

As described in the section, Pointers and the *based* Declarator, the address function is an extension to ALGOL that permits variable addressing on a level comparable to assembly language programming. Refer to that section for further information on use of pointers and *based* variables with the address function.

Example:

```
begin pointer p; integer array b;
      integer i; based integer x;
              :
      p :=address (b[i]);          ←pointer p is assigned the ad-
              :                     dress of array element b[i].
```

The address function permits parameter substitution by name and by value as do other function references:

```
begin pointer a; integer array b; integer i; based integer x;
      :
a :=address ((b[i]));
```

Length Function

The length function returns as a value the length of its character string argument. The function has the form:

```
length (v)
```

where: v is a string variable

Examples:

```
string (10) xyz;
xyz := 'abcd';
i :=length (xyz);          ←The assignment is the same as i :=4;
```

```
string (20) a,b;
a := 'abcdefg';
b :='xxx';
a(length(a)+1) :=b;        ←The subscript is evaluated and b concate-
                            nated with a.  The assignment is the same
                            as a := 'abcdefgxxx';
```

Index Function

The index function searches a specified character string for a specified character configuration. The function returns the starting location of the character configuration as its value. The function has the form:

```
index (v, c)
```

where: v is a string variable
c is one or more characters of v

Examples:

```
string (10) v;

v :='abcdefg';
i :=index (v,'bc');        ←The assignment is the same as i :=2;
    :
v :='abcdefg';
i :=index (v,'b');         ←The assignment is the same as i :=2;
```

Index Function (continued)

```
string (100) a, b;

a :=`abcdefg´;
b :=`xxx´;
a (index (a,`c´)) :=b;     ←The assignment replaces cde by xxx. It is
                            the same as a :=`abxxxfg´;
```

Shift Function

The shift function permits contents of a location to be shifted left or right. The function has the form:

```
shift (v,n)
```

where: v is an integer variable or octal literal.

n is a signed or unsigned integer.

The integer n indicates the number of bits to be displaced. A negative integer indicates left shift, and a positive or unsigned integer indicates right shift. Example:

```
x :=shift (x,+4);          ←right shift by 4 bits the contents of x.
```

Rotate Function

The rotate function permits contents of a location to be rotated left or right. The function has the form:

```
rotate (v,n)
```

where: v is an integer variable or octal literal.

n is an integer.

The integer n indicates the number of bits to be displaced. A negative integer indicates left rotate, and a positive or unsigned integer indicates right rotate. Example:

```
i :=rotate (x,-4);         ←value stored in i is contents of x left
                            rotated by four bits.
```

Tab Function

The tab function causes horizontal tabulation of the teletypewriter carriage. The function has the form:

```
tab (n)
```

where: n is an integer variable or unsigned integer

The integer n indicates the number of the column to which the carriage will move. If n is smaller than the current carriage column, line feed will occur before shifting to the indicated column.

Example:

```
output (1, "###", a, tab (10), b, tab (20));
```

Page Function

The page function causes a teletypewriter form feed. The function has the form:

```
page
```

Paging is usually made by value with the function name enclosed in parentheses.

```
output (1, "#####.##", a[i,j]);
write  (1, (page));
```

I/O RUNTIME PROCEDURES

Since standard ALGOL was designed to be a language independent of specific processors or devices, no I/O statements or conventions are included in the ALGOL specification.

For user convenience, five external procedures are available with NOVA ALGOL to handle I/O operations. The procedures are simply runtime routines that can be called by a user program using a procedure statement.

If the user wishes, he can implement additional I/O features by writing his own external procedures to handle input and output.

Open a File

Call Format:

```
open (fileno, string);
```

where: fileno is one of the channels (0-7) that can be associated with a given file.

string is the character string giving the file name.

Purpose: The procedure is called to open a file for input or output.

Example:

```
begin procedure xyz;
        .
        .
open (2,'infile1');
        .
        .
open (3,'testproc');
        .
        .
        .
```

Read a File

Call Format:

```
read(fileno, list);
```

where: fileno is the number of a channel (0-7) associated with the file.

list is a list of input data.

Purpose: The procedure is called to input data from a file.

Input Data: Input data may be numbers (1, 2, 5E10, etc.), or strings. Strings may have the formats:

```
"xxx...xxx" or yyy...yyyd
```

where: each x is any character.
d is one of the delimiters: space, comma, semicolon, CR, or TAB.

each y is any character except one of the delimiters.

Examples:

```
read (2, name1, name2, socsec1, socsec2, socsec3);
```

The file contains: John Jones 117 33 9666

```
read (2, `John Jones´, socsec1, socsec2, socsec3);
```

The file is the same as shown before.

Write a File

Call Format:

```
write (fileno, list);
```

where: fileno is the channel number (0-7) associated with the file.

list is a list of variables and constants to be written out.

Purpose: The procedure is called to write data from a file to an output device.

Write a File (continued)

Data List: Variable data format depends upon the type of datum and does not need to be specified.

An unsubscripted array name in the list causes output of all array elements.

Example:

```
write (1, x[5,3], y, sub);
```

Data represented by variables x[5,3], y, and sub are written to file number 1.

Close a File

Call Format:

```
close (fileno);
```

where: fileno is the channel number (0-7) currently associated with a file through an open procedure.

Purpose: The procedure is called to close a file after I/O is completed.

Example:

```
close (1);
```

The file previously associated with channel number 1 is closed.

Formatted Output

Call Format:

```
output (fileno, format, variable-list)
```

where: fileno is the channel number associated with the file (0-7).

format is a specification of output format.

variable list is a list of variables to be written out according to the given format. Carriage control functions can be included in the variable list.

Formatted Output (continued)

The specification of format is enclosed in either accent marks or quotation marks. The specification may be a string that is output precisely as given in the specification.

```
output (1, "Data Reduction");         ←procedure call

Data Reduction                        ←resultant output
```

A decimal point can be part of a field format. Assume values for the variables as follows: w=1.234, x=-99, y=.09, z=99999.9.

```
output (2, "#####.# ",w,x,y,z);    ←procedure call

    1.2   -99.0      .1 99999.9    ←output.  Note rounding.
```

If a number of values are to be written on a given line, the output procedure will put one blank character between the values even if the field format does not include extra character positions.

The field format can be unsigned or signed (+ or -).

If the field is unsigned, the sign is typed only for negative numbers and occupies a field position. For example the range of field ###.## would be -99.99 to 999.99.

If the field is signed, the sign does not require a field position. For example the range of field +###.## or -###.## would be -999.99 to +999.999.

A negative field sign causes printout only of negative signs. A positive field sign causes printout of both negative and positive signs.

Results of outputting values with or without signs are compared on the next page for the following values: g=-222.22, h=45.45, i=-3333.44. Blank field positions are shown by a triangle.

Formatted Output (continued)

```
output (2, "#####.## ",g,h,i);
output (2, "-#####.##",g,h,i);
output (2, "+#####.##",g,h,i);

Δ-222.22ΔΔΔΔΔ45.45ΔΔ-3333.44        ←unsigned
ΔΔ-222.22ΔΔΔΔΔ45.45ΔΔΔ-3333.44      ←negative
ΔΔ-222.22ΔΔΔΔΔ+45.45ΔΔΔ-3333.44     ←positive
```

A 3-position exponent field is allowed as part of a decimal field, using the symbol ↑.

```
output (2, "-#####.##↑↑↑ ", w,x,y,z);←values shown on the previous page

 12340.00-04     -99.00+00  90000.00-06  99999.90+00
```

Strings are represented by the symbol # for each character. Output is left justified in the field with trailing blanks if necessary.

The length of the string in characters should not exceed the number of field format positions. If it does, the extra characters will be written over the next field.

```
output (2, "######## ",g,h);          ←procedure call

TITLE    NUMBER                       ←possible output
```

A field format can be used for a string value given in the output call. The string is left justified in the field with trailing blanks if necessary.

```
output (2, "########", g,h, "YEAR"); ←procedure call

TITLE   NUMBER  YEAR                  ←possible output
```

Any field format may be used for a string value given in an output call.

```
output (3, "####.##↑↑↑", "YEAR");     ←procedure call

YEAR                                  ←output
```

Formatted Output (continued)

The carriage control functions, page and tab, can be incorporated into the variable list following the specification. The page function should be enclosed in parentheses to insure a by value call.

```
output (2, "#####.##", a, tab (15), b, tab (30), c, tab (45));

 4678.23      -234.40         678.49         -233.43
```

```
output (2, "####" a, tab (15), b, tab (0), c, tab (15), d);

4678          -234
 678          -233
```

An array identifier in a variable list causes all elements of the array to be written out in normal array sequence.

```
output (2, "####", A);          ←A is an array identifier

  34 5781  777 1234  354   9   100 4555 9000  888 ←possible output
```

Loops can be set up using output procedure calls, making it possible to produce output data in a number of formats.

```
for j :=1 step 5 until 100 do
output (2, "#### "  , a[j], a[j+1], a[j+2], a[j+3], a[j+4]);

   0  1020  4545  6123  9081
1003  2354   765  9034  7777     Part of possible output
3334  9654  3030  7077  8451
```

```
begin  literal  s(" A[###] = ####  ####  ####  ####  ####");
based  integer  array ba[0:4]; pointer p;

for j :=1 step 5 until 100 do
begin p :=address (a[j]);
      output (2,s,j,p→ba);
      end;
```

Part of the possible output might be:

```
A[  1] = 1005  1195  3142  5222  1110
A[  6] = 2019  3001  4100  1955  5111
A[ 11] = 2211  3521  4321  1236  6000
```

ALGOL ERROR MESSAGES

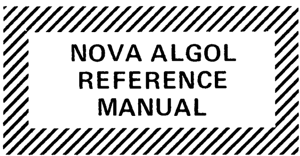
"ER nn" ←compiler error - notify Data General Corporation.
"identifier is not a label" ←label required.
"illegal type declaration"
"illegal syntax" ←message commonly resulting from programming errors.
"illegal declaration"
"missing variable in expression"
"missing operator in expression"
"illegal operator in expression"
"illegal use of reserved word"
"illegal symbol in expression" e.g., (a + b) :=0
"illegal operator" e.g., a := a *for* b;
"undefined variable"
"boolean in real expression"
"illegal pointer"
"illegal use of a string"
"no subscripts for array reference"
"wring number of subscripts"
"expression does not end properly"
"parentheses do not balance"
"illegal subscript end" e.g., a[b + (CC**]
"function reference does not end properly"
"illegal parameter separator"
"wrong type for operator" e.g., *real* x; x *and* 0
"illegal variable"
"illegal type conversion"
"illegal character"
"illegal character in comment"
"illegal string literal"
"illegal number"
"illegal function reference"
"duplicate symbol definition"
"illegal use of an operator"
"illegal character in number"
"precision must be an integer number"
"real number is out of range"
"assigned local storage full"
"more than one decimal point"
"compiler table overflow"

All error messages are printed out as in the example:

```
a :=b/*c;
     ↑
***missing variable in expression ***
```

The arrow indicates where the error was discovered, not necessarily the location of the error. There are also several other I/O dependent messages, e.g., in the monitor version:

```
"input file does not exist"
```



EXTENSIONS TO STANDARD ALGOL

external procedures and *external* variables.

Character string variables and arrays. String manipulation using built-in index and length functions and substring capability.

Bit manipulation using octal and binary literals and the built-in shift and rotate functions.

xor (exclusive or) Boolean operator.

I/O routines providing free form read and write or formatted output.

based and *pointer* variables and built-in address function for efficient addressing.

Subscripted labels.

Built-in carriage control functions, tab and page.

Data type conversions:

integer to *boolean* and *boolean* to *integer*
integer to *pointer* and *pointer* to *integer*
integer to *string*

Complex data declared with the *complex* data type.

Conversion to any radix up to and including 10.

Inclusion of strings as list items of the *for* statement.

Declaration of an identifier as *literal* with a given value to be substituted in statements of the program.

LIMITATIONS OF NOVA ALGOL

NOVA ALGOL implements the complete standard ALGOL with the following minor exceptions:

No blanks within identifiers are permitted. An underscore (_) may be used to separate logical parts of identifiers.

Data types must be declared for all parameters.

The separator)*string*:(is allowed only in the headings of procedures.

Identifiers may not contain more than 32 characters.

ALGOL keywords (see page 1-1) may not be redefined as program identifiers.

APPENDIX A

CALLS AND RETURNS

When actual parameters replace formal parameters, certain information about the actual parameter must be passed with the identifier. A single 16-bit word, called a specifier contains this information as shown:

0 shape	4 storage class	8 data type	12 precision 15

The following key indicates the specifier contents:

Shape

0 scalar
1 array
2 procedure

Storage Class

0 local
1 own
2 parameter
3 based
4 value
5 external

Data Type

0 undefined
1 integer
2 real
3 boolean
4 label
5 complex
6 pointer
7 multi-precision integer
8 string

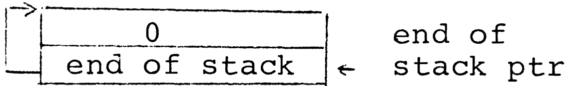
A stack frame is set up to provide return information. It has the following format:

	Stack frame	
.SP (stack pointer) →	return address	
fixed-length information	AC0	
	AC1	
	AC2	
	old stack pointer	→ to previous stack frame
	end of stack pointer	→ points to end of frame
	carry and level info	
	assigned (compile-time) local area	
	allocated (run-time) local area	

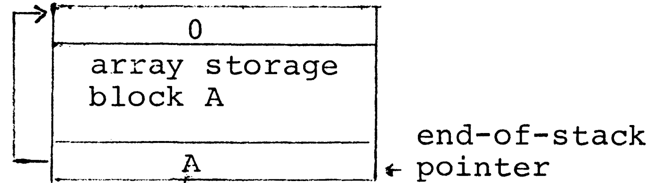
CALLS AND RETURNS (continued)

Within the allocated area, backward and forward threaded block pointers are used. This assures that a proper return can be made through any number of block levels. A series of views of changes in the allocated local area will show how the threaded block pointers work.

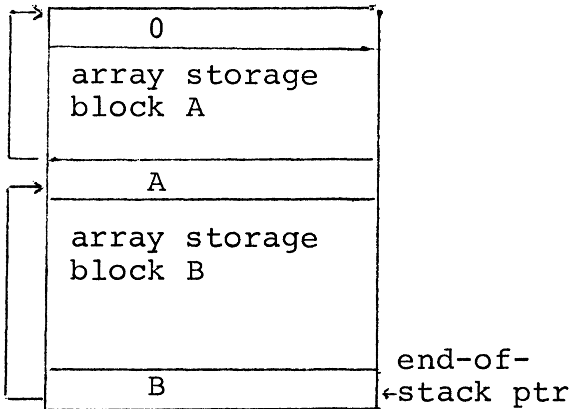
1. Nothing in Allocated area



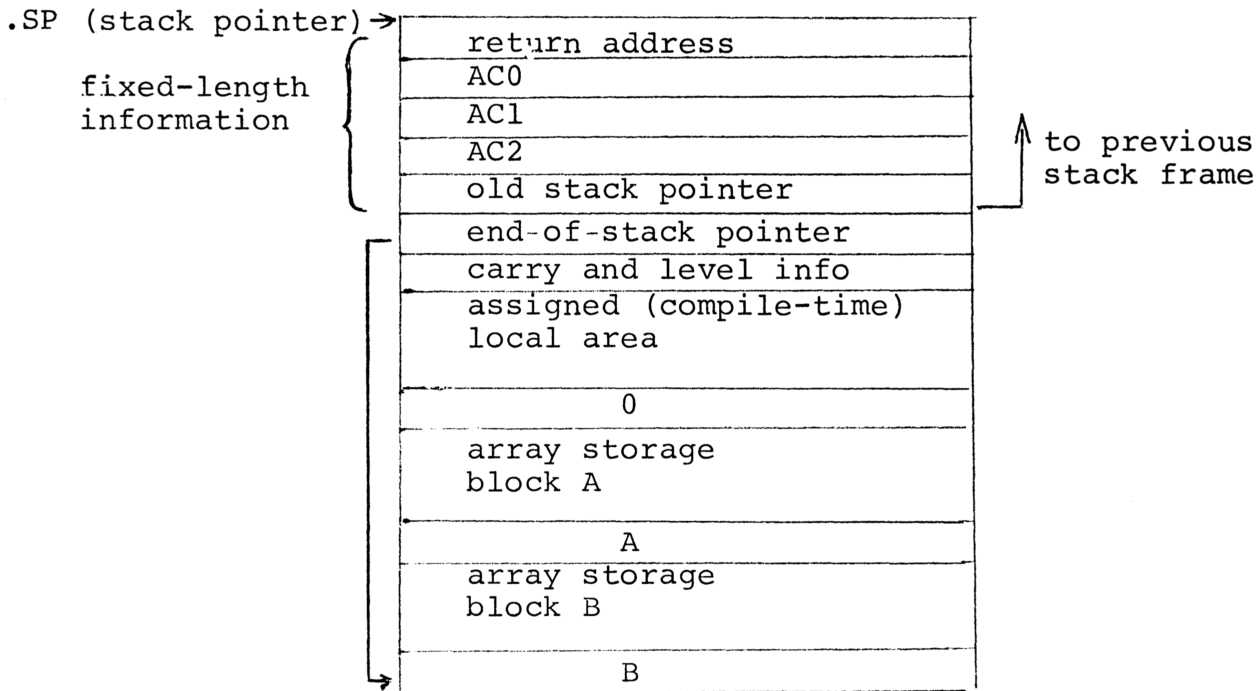
2. Array of Block A added



3. Array of Block B Added



The complete stack frame would than appear as:



CALLS AND RETURNS (continued)

The following table shows stack frame allocation according to shape, storage class, and data type.

	STORAGE CLASS						
SCALARS	local	own	para- meter	based	value	exter- nal	PROCE- DURES
DATA TYPE							
Integer	AL 1	AO 1	---	---	AL 1	---	AL 1
Real	AL 2 or P	AO 2 or P	---	---	AL 2 or P	--- 2 or P	AL 2 or P
Boolean	AL 1	AO 1	---	---	AL 1	---	AL 1
Label	illegal 1	illegal 1	---	illegal	illegal 1	---	AL 1
Complex	AL 4 or P	AO 4 or P	---	---	AL 4 or P	---	AL,SL 2
Pointer	AL 2	AO 2	---	---	AL 2	---	AL 2
Multi-Prec. Integer	AL 1 or P	AO 1 or P	---	---	AL 1 or P	---	AL 1 or P
String	AL,SL 2	AO,SO 2	---	---	AL,SL 2	---	AL,SL 2
ARRAYS							
Integer	AL,SL 1	AO,SO 2	---	AL,SL* 2	AL,SL 2	---	
Real	AL,SL 2	AO,SO 2	--- ---	AL,SL* 2	AL,SL 2	---	
Boolean	AL,SL 2	AO,SO 2	---	AL,SL* 2	AL,SL 2	---	
Label	illegal	illegal	illegal	illegal	illegal	---	
Complex	AL,SL 4	AO,SO 4	---	AL,SL* 4	AL,SL 4	---	
Pointer	AL,SL 2	AO,SO 2	--- 2	AL,SL* 2	AL,SL 2	---	
Multi-Prec. Integer	AL,SL 2	AO,SO 2	---	AL,SL* 2	AL,SL 2	---	
String	AL,SL 2	AO,SO 2	---	---	AL,SL 2	---	

Key:

AL = assigned local
SL = allocated local
AO = assigned own
SO = allocated own
P = defined precision
* = note that specifier is allocated
--- = no storage allocated

APPENDIX B

RUN-TIME ROUTINES

Run-time routines, used to maintain stacks described in APPENDIX A, are compatible with NOVA assembly language. The coding and use of the routines are described below.

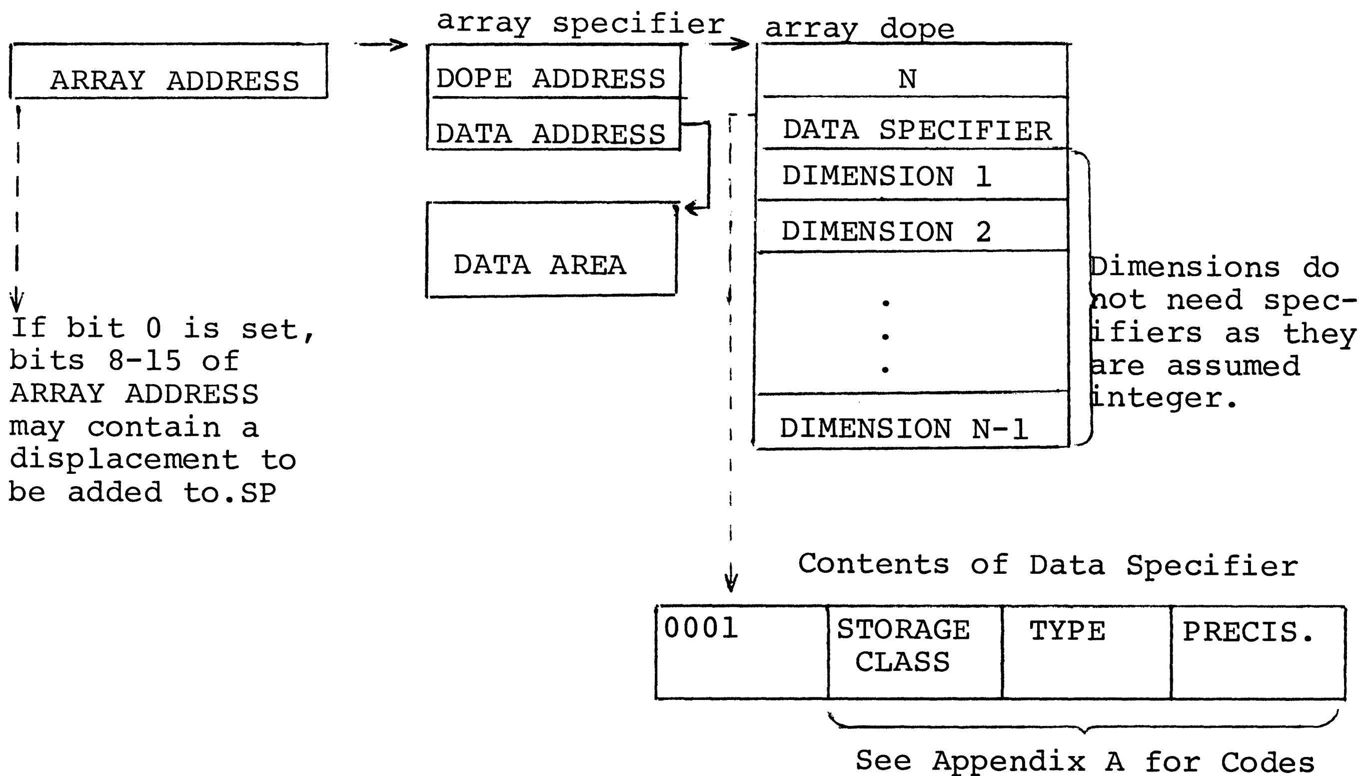
ROUTINE	CODING	MEANING
CALL	JSR @CALL SUBR	SUBR is the label of the program to called. CALL stores JSR+2 in the stack frame return location for RETURN, sets AC3=.SP (stack pointer), and jumps to SUBR. See NOTE at bottom of page.
SAVE	JSR @SAVE N	N has the form: [level | words]. SAVE builds a new stack frame. SAVE sets the end of stack and stack level, checks for stack overflow, and sets the old stack pointer. SAVE saves AC1→AC3, and Carry, sets AC3=.SP, and jumps to JSR+2.
RETURN	JSR @RETURN	RETURN destroys the current stack frame, restoring AC1→AC3, and Carry. The previous stack is restored, and RETURN checks for top of stack. RETURN sets AC3=.SP and jumps to the stack return of the new stack. See NOTE at bottom of page.
JUMP	JSR @JUMP LOC	LOC is the address of the word to transfer to. JUMP sets AC3=.SP and jumps to LOC.
JUMP1	JSR @JUMP1	AC1=address of the word to transfer to. JUMP1 sets AC3=.SP and jumps to AC1 location.
JUMPC	JSR @JUMPC LOC	JUMPC sets AC3=.SP and jumps to LOC if the Carry bit is set.

NOTE: The short form of a program call and return is:

```
        JSR    SUBR        ;CALL
SUBR:   STA    3,@.SP      ;SAVE RETURN
        JMP    @.SP        ;RETURN
```

Run-Time Routines (continued)

ROUTINE	CODING	MEANING
ARRAY	JSR @ARRAY N ARRAY ADDRESS ARRAY SPECIFIER DIMENSION 1 ADDR. DIMENSION 2 ADDR. . . . DIMENSION N-1 ADDR.	ARRAY ADDRESS is a one-word pointer to a two-word array specifier. See the illustration below for the meaning of the various fields. ARRAY sets up the two-word specifier, builds the control table, calculates the data area needed, and adjusts the end of stack. While doing so, ARRAY performs a series of checks on dimensions, specifier, and the stack. All registers are destroyed, and AC3=.SP.



Run-Time Routines (continued)

ROUTINE	CODING	MEANING
SUBSCRIPT	JSR @SUBSCRIPT N COMPUTED ADDRESS ARRAY ADDRESS SUBSCRIPT 1 SUBSCRIPT 2 . . . SUBSCRIPT N-1	The COMPUTED ADDRESS is an address of a word containing the address of data. SUBSCRIPT computes the address of the data, using the algorithm given below.

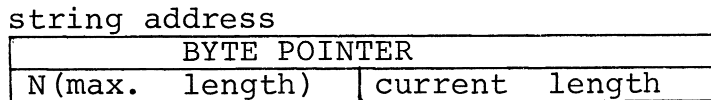
Subscript Algorithm

$$(sub1-1)\times(dim2\times dim3\times\ldots\times dim_{last})+(sub2-1)\times(dim3\times dim4\times\ldots\times dim_{last})+\ldots+(subI-1)\times(dim(I-1)\times dim(I-2)\times\ldots\times dim_{last})+\ldots+sub_{last}$$

where: dim(I) = dimB-dimA+1

ROUTINE	CODING	MEANING
GETSP	JSR @GETSP L	L is the level desired. GETSP searches for the desired level until it is found or until level 0 is found. GETSP places the level .SP in AC3. A check is made for: next level - last level =0 or 1
BLKSTART	JSR @BLKSTART	BLKSTART places an indirect bit at the end of stack (-1).
BLKEND	JSR @BLKEND	BLKEND searches through the threads at the stack end for the indirect bit and sets it as the current end of stack, which destroys the current block.
STRING	JSR @STRING STRING ADDRESS N	N is the maximum number of characters of string. STRING builds a pointer at STRING ADDRESS as shown:

string address



BYTE POINTER	
N(max. length)	current length

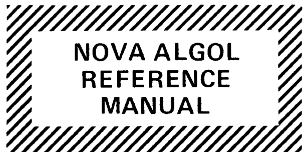
STRING then adjusts the end of stack adding N/2 to stack length.

Run-Time Routines (continued)

ROUTINE	CODING		MEANING
ALLOC	JSR N LOC	@ALLOC	ALLOC allocates N words in the *own* stack and stores the stack address in LOC.
LENGTH	JSR LOC1 LOC2	@ LENGTH	LOC1 is the string specifier address. LOC2 is the address containing the length of the string. LENGTH stores in LOC2 the current length of the string at LOC1 and sets AC3=.SP.
SUBSTR	JSR LOC1 LOC2 LOC3 LOC4	@SUBSTR	LOC1 is the string specifier address. LOC2 contains the number of the starting character. LOC3 contains the number of the terminal character. LOC4 is a temporary for new string data. SUBSTR makes a new string by subsetting an existing string from the character whose position is LOC2 to the character whose position is LOC3. Pointers are set in LOC4 as follows:

STARTING ADDRESS POINTER		0 1
Maximum Length L2-L1+1	Current Length Lcur-L1+1	

ROUTINE	CODING		MEANING
MOVESTR	JSR LOC1 LOC2	@MOVESTR	LOC1 and LOC2 are string specifier pointers. MOVESTR copies a string. The routine moves the smallest maximum length selected from LOC1 and LOC2 to LOC2.
STREQ	JSR LOC1 LOC2	@STREQ	LOC1 and LOC2 are string specifier pointers. STREQ sets AC0=0 if LOC1=LOC2.
STRCMP	JSR LOC1 LOC2	@STRCMP	LOC1 and LOC2 are string specifier pointers. STRCMP compares LOC1 and LOC2 and sets AC0 as follows: AC0=0 if LOC1=LOC2 AC0=1 if LOC1>LOC2 AC0=2 if LOC1<LOC2



APPENDIX C

LOADING THE ALGOL COMPILER

Stand-alone ALGOL compilers are available in 8K and 12K versions. NOVA ALGOL is a two-pass compiler, having the following tapes:

12K Version	8K Version
Pass 1 - 091-000030-00	Pass 1 - 091-000034-00
Pass 2 - 091-000031-00	Pass 2 - 091-000035-00

There are three device modes for loading ALGOL. All are possible alternatives for users having the 12K version of the compiler. Only modes 2 and 3 can be used with the 8K version.

The user should exercise care in loading the ALGOL compiler. Loading will take some time and, if interrupted, loading must be restarted from the beginning.

The device mode is established at the start of Pass 1 and cannot be altered during the compilation. The device modes are:

1	2	3	
TTI*	TTR	PTR	(Pass 1 input)
TTO	TTP	PTP	(Symbolic output)
PTR	TTR	PTR	(Intermediate input)
PTP	TTP	PTP	(Intermediate output)

In mode 3, as well as the other modes, the teletypewriter is required for system prompts and user responses.

At the start of loading the following prompt is issued:

MODE:

The user responds 1, 2 or 3, as appropriate. The number is followed by a carriage return. The ALGOL source is read from the Pass 1 input device, and the intermediate tape is punched out. The following message is then typed:

LOAD ALGOL PHASE 2
HIT 'RETURN' WHEN READY

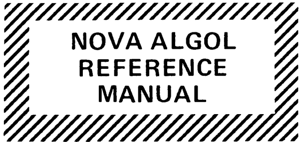
* Note that in Mode 1, the user can edit and correct input using the standard NOVA erase and kill characters:
← (RUBOUT key) - erase previous character
\ (SHIFT L keys) - delete the entire line

LOADING THE ALGOL COMPILER
(continued)

The user loads Pass 2 into the reader and hits carriage return to load. When the tape is read in, the following prompt is given:

LOAD INTERMEDIATE TAPE
HIT 'RETURN' WHEN READY

The user loads the intermediate tape into the appropriate device and then gives a carriage return. When the tape is loaded, results of compilation of the source code are output.



APPENDIX D

EXAMPLE OF ALGOL GENERATED CODE

Following are two versions of Algorithm 347 by Richard C. Singleton, which was published in the "Communications of the ACM", Volume 12, Number 3, March 1969.

The first example shows the source code for the algorithm as it was originally written.

In the second example source code and coding generated by Data General's ALGOL Compiler are shown for the same program when optimized with use of pointers and based variables.

```
PROCEDURE SORT(A, I, J);
        VALUE I, J;
        INTEGER I, J;
        REAL ARRAY A;

COMMENT:  THIS PROCEDURE SORTS THE ELEMENTS OF ARRAY A
        INTO ASCENDING ORDER, SO THAT

        A[K]=<A[K+1],    K=I,I+1,...,J-1.

        ALGORITHM 347, COMM. ACM, MARCH 1969.

        THIS ALGORITHM PROVIDES AN EFFICIENT METHOD FOR SORTING
        WITH MINIMAL STORAGE. IT IS THE FASTEST OF SEVERAL
        ALGORITHMS PUBLISHED IN COMM. ACM INCLUDING SCOWEN'S
        "QUICKERSORT" AND HOARE'S "QUICKSORT".;

BEGIN
        REAL T, TT;
        INTEGER II, IJ, K, L, M;
        INTEGER ARRAY IL, IU[Ø:LN(J-I+1)];

        M:=Ø;
        II:=I;
        GO TO 4;

1:      IJ:=(I+J)/2;
        T:=A[IJ];
        K:=I;
        L:=J;
        IF A[I]>T THEN BEGIN
                A[IJ]:=A[I];
                A[I]:=T;
                T:=A[IJ];
                END;
        IF A[J]<T THEN BEGIN
                A[IJ]:=A[J];
                A[J]:=T;
                T:=A[IJ];
                IF A[I]>T THEN BEGIN
                        A[IJ]:=A[I];
                        A[I]:=T;
                        T:=A[IJ];
                        END
                END;

2:      L:=L-1;
        IF A[L]>T THEN GO TO 2;
        TT:=A[L];
```

```
3:      K:=K+1;
        IF A[L]>T THEN GO TO 3;
        IF K=<L THEN BEGIN
                A[L]:=A[K];
                A[K]:=TT;
                GO TO 2;
                END;
        IF L-I>J-K THEN BEGIN
                IL[M]:=I;
                IU[M]:=L;
                I:=K;
                END


        ELSE BEGIN
                IL[M]:=K;
                IU[M]:=J;
                J:=L;
                END;
        M:=M+1;

4:      IF J-I>10 THEN GO TO 1;
        IF I=II THEN BEGIN
                IF I<J THEN GO TO 1;
                END;
        FOR I:=I+1 STEP 1 UNTIL J DO BEGIN
                T:=A[I];
                K:=I-1;
                IF A[K]>T THEN BEGIN
5:                      A[K+1]:=A[K];
                        IF A[K]>T THEN GO TO 5;
                        A[K+1]:=T;
                        END
                END;
        M:=M-1;
        IF M>0 THEN BEGIN
                I:=IL[M];
                J:=IU[M];
                GO TO 4;
                END

        END SORT
```

```
PROCEDURE SORT(A, I, J);
        VALUE I, J;
        INTEGER I, J;
        POINTER A;

COMMENT:  THIS PROCEDURE SORTS THE ELEMENTS OF ARRAY A
        INTO ASCENDING ORDER, SO THAT

        A[K]=<A[K+1],    K=I,I+1,...,J-1.

        ALGORITHM 347, COMM. ACM, MARCH 1969.

        THIS ALGORITHM PROVIDES AN EFFICIENT METHOD FOR SORTING
        WITH MINIMAL STORAGE.  IT IS THE FASTEST OF SEVERAL
        ALGORITHMS PUBLISHED IN COMM. ACM INCLUDING SCOWEN'S
        "QUICKERSORT" ALGORITHM AND HOARE'S "QUICKSORT".;

BEGIN
        INTEGER II, IJ, K, L, M, TT, T;
        EXTERNAL POINTER IL, IU;
        POINTER AI, AJ, AIJ, AK, AL, AK1, ILM, IUM;
        BASED INTEGER N;

        M := 0;
        II := I;
        GO TO 4;

1:      IJ := (I+J)/2;
        AIJ := A+IJ;
        T := AIJ->N;
        AK := AI := A+I;
        AL := AJ := A+J;
        K := I;
        L := J;
        IF AI->N>T THEN BEGIN
                AIJ->N := AI->N;
                AI->N := T;
                T := AIJ->N;
                END;
        IF AJ->N<T THEN BEGIN
                AIJ->N := AJ->N;
                AJ->N := T;
                T := AIJ->N;
                IF AI->N>T THEN BEGIN
                        AIJ->N := AI->N;
                        AI->N := T;
                        T := AIJ->N;
                        END
                END;

2:      L := L-1;
        AL := A+L;
        IF AL->N>T THEN GO TO 2;
        TT := AL->N;
```

```
3:      K := K+1;
        AK := A+K;
        ILM := IL+M;
        IUM := IU+M;
        IF AL->N>T THEN GO TO 3;
        IF K=<L THEN BEGIN
                AL->N := AK->N;



                AK->N := TT;
                GO TO 2;
                END;
        IF L-I>J-K THEN BEGIN
                ILM->N := I;
                IUM->N := L;
                I := K;
                END
        ELSE BEGIN
                ILM->N := K;
                IUM->N := J;
                J := L;
                END;
        M := M+1;

4:      IF J-I>10 THEN GO TO 1;
        IF I=II THEN BEGIN
                IF I<J THEN GO TO 1;
                END;
        FOR I := I+1 STEP 1 UNTIL J DO BEGIN
                T := (A+I)->N;
                K := I-1;
                AK := A+K;
                AK1 := AK+1;
                IF AK->N>T THEN BEGIN
5:                      AK1->N := AK->N;
                        IF AK->N>T THEN GO TO 5;
                        AK1->N := T;
                        END
                END;
        M := M-1;
        IF M>0 THEN BEGIN
                I := (IL+M)->N;
                J := (IU+M)->N;
                GO TO 4;
                END

        END SORT
```

```
; PROCEDURE SORT(A, I, J);

        .EXTU
        .ENT    SORT

        .ZREL
.LP:    LP

        .NREL
SORT:   JSR     @SAVE
        FS1
        4
        SP+2                    ;A
        001141                  ;POINTER PARAMETER
        SP+0                    ;I
        002021                  ;INTEGER VALUE
        SP+1                    ;J
        002021                  ;INTEGER VALUE

;       VALUE I, J;
;       INTEGER I, J;
;       POINTER A;
;
; COMMENT:  THIS PROCEDURE SORTS THE ELEMENTS OF ARRAY A
;       INTO ASCENDING ORDER, SO THAT
;
;       A[K]=<A[K+1],    K=I,I+1,...,J-1.
;
;       ALGORITHM 347, COMM. ACM, MARCH 1969.
;
;       THIS ALGORITHM PROVIDES AN EFFICIENT METHOD FOR SORTING
;       WITH MINIMAL STORAGE.  IT IS THE FASTEST OF SEVERAL
;       ALGORITHMS PUBLISHED IN COMM. ACM INCLUDING SCOWEN'S
;       "QUICKERSORT" ALGORITHM AND HOARE'S "QUICKSORT".;
;
; BEGIN
;       INTEGER II, IJ, K, L, M, TT, T;
;       EXTERNAL POINTER IL, IU;
;       POINTER AI, AJ, AIJ, AK, AL, AK1, ILM, IUM;
;       BASED INTEGER N;
;
;       M := 0;

        SUB     0,0             ;(0)
        LDA     3,.SP
        STA     0,S+7,3         ;M

;       II := I;

        LDA     1,S+0,3         ;I
        STA     1,S+3,3         ;II
```

```
;           GO TO 4;

            JSR       @JUMP
            L4

;
; 1:        IJ := (I+J)/2;

L1:         LDA       3,.SP
            LDA       Ø,S+Ø,3             ;I



            LDA       1,S+1,3             ;J
            ADD       Ø,1
            LDA       3,.LP
            LDA       2,1,3               ;LITERAL
            SUBO      Ø,Ø
            JSR       @SDIV               ;SIGNED DIVIDE
            LDA       3,.SP
            STA       1,S+4,3             ;IJ

;           AIJ := A+IJ;

            LDA       Ø,S+2,3             ;A
            ADD       Ø,1
            STA       1,S+14,3            ;AIJ

;           T := AIJ->N;

            MOV       1,2
            LDA       1,Ø,2
            STA       1,S+11,3            ;T

;           AK := AI := A+I;

            LDA       1,S+Ø,3             ;I
            ADD       Ø,1
            STA       1,S+12,3            ;AI
            STA       1,S+15,3            ;AK

;           AL := AJ := A+J;

            LDA       1,S+1,3             ;J
            ADD       Ø,1
            STA       1,S+13,3            ;AJ
            STA       1,S+16,3            ;AL

;           K := I;

            LDA       Ø,S+Ø,3             ;I
            STA       Ø,S+5,3             ;K
```

```
;       L := J;

        LDA     0,S+1,3         ;J
        STA     0,S+6,3         ;L

;       IF AI->N>T THEN BEGIN

        LDA     0,@S+12,3       ;AI
        LDA     1,S+11,3        ;T
        ADCL    1,0
        SUBL    0,0
        MOVR    0,0
        JSR     @JUMPC
        G1

;               AIJ->N := AI->N;

        LDA     0,@S+12,3       ;AI
        STA     0,@S+14,3       ;AIJ

;               AI->N := T;

        LDA     1,S+11,3        ;T



        STA     1,@S+12,3       ;AI

;               T := AIJ->N;
;               END;

        STA     0,S+11,3        ;T

;       IF AJ->N<T THEN BEGIN

G1:     LDA     3,,SP
        LDA     0,S+11,3        ;T
        LDA     1,@S+13,3       ;AJ
        ADCL    1,0
        SUBL    0,0
        MOVR    0,0
        JSR     @JUMPC
        G2

;               AIJ->N := AJ->N;

        LDA     0,@S+13,3       ;AJ
        STA     0,@S+14,3       ;AIJ

;               AJ->N := T;

        LDA     1,S+11,3        ;T
        STA     1,@S+13,3       ;AJ
```

```
;               T := AIJ->N;
;               IF AI->N>T THEN BEGIN

        STA     0,S+11,3        ;T
        LDA     2,@S+12,3       ;AI
        ADCL    0,2
        SUBL    2,2
        MOVR    2,2
        JSR     @JUMPC
        G3

;                       AIJ->N := AI->N;

        LDA     0,@S+12,3       ;AI
        STA     0,@S+14,3       ;AIJ

;                       AI->N := T;

        LDA     1,S+11,3        ;T
        STA     1,@S+12,3       ;AI

;                       T := AIJ->N;
;                       END
;               END;

        STA     0,S+11,3        ;T
G3:

;
; 2:    L := L-1;

G2:
L2:     LDA     3,.SP
        LDA     0,S+6,3         ;L
        SUBZL   1,1             ;(1)



        NEG     1,1
        ADD     0,1
        STA     1,S+6,3         ;L

;       AL := A+L;

        LDA     0,S+2,3         ;A
        ADD     0,1
        STA     1,S+15,3        ;AL
```

```
;         IF AL->N>T THEN GO TO 2;

          MOV       1,2
          LDA       1,0,2
          LDA       0,S+11,3            ;T
          ADCL      0,1
          SUBL      1,1
          MOVR      1,1
          JSR       @JUMPC
          G4
          JSR       @JUMP
          L2

;         TT := AL->N;

G4:       LDA       3,.SP
          LDA       0,@S+16,3           ;AL
          STA       0,S+10,3            ;TT

;
; 3:      K := K+1;

L3:       LDA       3,.SP
          LDA       0,S+5,3             ;K
          INC       0,0
          STA       0,S+5,3             ;K

;         AK := A+K;

          LDA       1,S+2,3             ;A
          ADD       1,0
          STA       0,S+15,3            ;AK

;         ILM := IL+M;

          LDA       2,IL                ;IL
          LDA       0,S+7,3             ;M
          ADD       2,0
          STA       0,S+20,3            ;ILM

;         IUM := IU+M;

          LDA       0,IU                ;IU
          LDA       1,S+7,3             ;M
          ADD       0,1
          STA       1,S+21,3            ;IUM

;         IF AL->N>T THEN GO TO 3;

          LDA       0,@S+16,3           ;AL
          LDA       1,S+11,3            ;T
          ADCL      1,0
          SUBL      0,0
          MOVR      0,0
          JSR       @JUMPC
          G5
          JSR       @JUMP
          L3
```

```
;        IF K=<L THEN BEGIN

G5:      LDA     3,.SP
         LDA     0,S+5,3          ;K
         LDA     1,S+6,3          ;L
         SUBL    0,1
         SUBL    1,1
         MOVR    1,1
         JSR     @JUMPC
         G6

;                AL->N := AK->N;

         LDA     0,@S+15,3        ;AK
         STA     0,@S+16,3        ;AL

;                AK->N := TT;

         LDA     1,S+10,3         ;TT
         STA     1,@S+15,3        ;AK

;                GO TO 2;

         JSR     @JUMP
         L2

;                END;
;        IF L-I>J-K THEN BEGIN

G6:      LDA     3,.SP
         LDA     0,S+6,3          ;L
         LDA     1,S+0,3          ;I
         NEG     1,1
         ADD     0,1
         LDA     2,S+1,3          ;J
         LDA     0,S+5,3          ;K
         NEG     0,0
         ADD     2,0
         SUBL    1,0
         SUBCL   0,0
         MOVR    0,0
         JSR     @JUMPC
         G7

;                ILM->N := I;

         LDA     0,S+0,3          ;I
         STA     0,@S+20,3        ;ILM

;                IUM->N := L;

         LDA     1,S+6,3          ;L
         STA     1,@S+21,3        ;IUM
```

```
;               I := K;

        LDA     2,S+5,3         ;K
        STA     2,S+0,3         ;I

;               END
;       ELSE BEGIN

        JSR     @JUMP
        G10

;               ILM->N := K;

G7:     LDA     3,.SP
        LDA     0,S+5,3         ;K
        STA     0,@S+20,3       ;ILM

;               IUM->N := J;

        LDA     1,S+1,3         ;J
        STA     1,@S+21,3       ;IUM

;               J := L;

        LDA     2,S+6,3         ;L
        STA     2,S+1,3         ;J

;               END;
;       M := M+1;

G10:    LDA     3,.SP
        LDA     0,S+7,3         ;M
        INC     0,0
        STA     0,S+7,3         ;M

;
; 4:    IF J-I>10 THEN GO TO 1;

L4:     LDA     3,.SP
        LDA     0,S+1,3         ;J
        LDA     1,S+0,3         ;I
        NEG     1,1
        ADD     0,1
        LDA     3,.LP
        LDA     2,3,3           ;LITERAL
        ADCL    2,1
        SUBL    1,1
        MOVR    1,1
        JSR     @JUMPC
        G11
        JSR     @JUMP
        L1
```

```
;       IF I=II THEN BEGIN

G11:    LDA     3,,SP
        LDA     0,S+0,3         ;I
        LDA     1,S+3,3         ;II
        SUB     0,1,SZR
        SUB     1,1,SKP
        INC     1,1
        MOVR    1,1
        JSR     @JUMPC
        G12

;               IF I<J THEN GO TO 1;

        LDA     0,S+1,3         ;J
        LDA     1,S+0,3         ;I
        ADCL    1,0
        SUBL    0,0
        MOVR    0,0
        JSR     @JUMPC
        G13
        JSR     @JUMP
        L1

;               END;

G13:

;       FOR I := I+1 STEP 1 UNTIL J DO BEGIN

G12:
G16:    LDA     3,,SP
        LDA     0,S+0,3         ;I
        INC     0,0
        STA     0,S+0,3         ;I
G17:    LDA     3,,SP
        LDA     0,S+0,3         ;I
        LDA     1,S+1,3         ;J
        SUBL    0,1
        MOV     0,0,SZC
        JMP     G20
        SUBZL   1,1             ;(1)
        ADD     1,0
        STA     0,S+0,3         ;I
        JSR     @.+1
        G15
        JSR     @JUMP
        G17
G20:    JSR     @JUMP
        G14
G15:    MOV     3,1             ;SAVE
```

```
;               T := (A+I)->N;

        LDA     3,,SP
        LDA     0,S+2,3         ;A
        LDA     2,S+0,3         ;I
        ADD     0,2
        LDA     2,0,2
        STA     2,S+11,3        ;T

;               K := I-1;

        LDA     0,S+0,3         ;I
        SUBZL   2,2             ;(1)
        NEG     2,2
        ADD     0,2
        STA     2,S+5,3         ;K

;               AK := A+K;

        LDA     0,S+2,3         ;A
        ADD     0,2
        STA     2,S+15,3        ;AK

;               AK1 := AK+1;

        INC     2,2
        STA     2,S+17,3        ;AK1

;               IF AK->N>T THEN BEGIN

        LDA     0,@S+15,3       ;AK
        LDA     2,S+11,3        ;T
        ADCL    2,0
        SUBL    0,0
        STA     1,S+23,3        ;TEMPORARY
        MOVR    0,0
        JSR     @JUMPC
        G21

; 5:                      AK1->N := AK->N;

L5:     LDA     3,,SP
        LDA     0,@S+15,3       ;AK
        STA     0,@S+17,3       ;AK1

;                         IF AK->N>T THEN GO TO 5;

        LDA     1,@S+15,3       ;AK
        LDA     2,S+11,3        ;T
        ADCL    2,1
        SUBL    1,1
        MOVR    1,1
        JSR     @JUMPC
        G22
        JSR     @JUMP
        L5
```

```
;                           AK1->N := T;

G22:    LDA     3,.SP
        LDA     0,S+11,3            ;T
        STA     0,@S+17,3           ;AK1

;                           END
;               END;

G21:    LDA     3,.SP
        LDA     1,S+23,3            ;RETURN
        JSR     @JUMP1

;       M := M-1;

G14:    LDA     3,.SP
        LDA     0,S+7,3             ;M
        SUBZL   1,1                 ;(1)
        NEG     1,1
        ADD     0,1
        STA     1,S+7,3             ;M

;       IF M>0 THEN BEGIN

        SUB     0,0                 ;(0)
        ADCL    0,1
        SUBL    1,1
        MOVR    1,1
        JSR     @JUMPC
        G23

;               I := (IL+M)->N;

        LDA     0,IL                ;IL
        LDA     1,S+7,3             ;M
        ADD     0,1
        MOV     1,2
        LDA     1,0,2
        STA     1,S+0,3             ;I

;               J := (IU+M)->N;

        LDA     2,IU                ;IU
        LDA     0,S+7,3             ;M
        ADD     2,0
        MOV     0,2
        LDA     0,0,2
        STA     0,S+1,3             ;J
```

```
;                  GO TO 4;

          JSR      @JUMP
          L4

;                  END
;
;         END SORT

G23:      JSR      @RETURN
FS1=      23

SP=       100000
LP:       000000
          000002
          000001
          000012

          .END
```

COMBINED INDEX

Subject	Page
+	4-1,4-4,4-8
-	4-1,4-4,4-8
×	4-1,4-4,4-8
/	4-1,4-4,4-8
↑	4-1,4-4,4-8
< ≤ = ≠ ≥ >	19,4-1,4-7,4-8
,	2,4-1,4-2
;	2,4-1,4-2
:	2,4-1,4-2
.	4-1,4-2
∧ ∨ ≡ ⊃ ¬ ⊕	19,4-1,4-7,4-8,4-9
_	1-1,12-1
10 (E,e)	4-1,4-2,4-5
:=	13,4-1,4-2,4-8,6-3
(space)	2,4-1,4-2
()	2,4-1,4-3
[]	2,4-1,4-3
` ' " "	32,4-1,4-3,7-6
→ ->	4-1,4-2,7-9ff
*	4-1
>= =>	4-1,4-7
<= =<	4-1,4-7
∧= <>	4-1,4-7
← (RUBOUT)	App. C
\ (SHIFT L)	App. C
abs built-in function	9-1
accent marks	32,4-1,4-3,7-6
actual parameters	25,8-2,8-5,8-6, App. A
addition	4-1,4-4,4-8
address built-in function	7-9 to 7-11,9-2
addressing by pointer	7-9 to 7-13
ALGOL	
calls and returns	App. A
error messages	11-1
extensions	12-1
limitations	12-1
loading of	App. C
run-time routines	App. B
versions	App. C
alloc run-time routine	App. B
allocation of storage	4,8 to 10, Chapter 2, Chapter 3, App. A
and	19,4-1,4-7,4-8,4-9

Subject	Page
arctan built-in function	9-1
arithmetic	
assignment	6-3
built-in functions	9-1
evaluation	4-4,4-8
expressions	5-1 to 5-3
numbers	4-5,4-6
operators	4-1,4-4
operator precedence	4-8
array declarator	5,4-1,7-1,7-3
array	
bounds	7,7-3
declaration	5,7-3
definition	5,7-3
dimensions	6,7-3
element	5,7-4
of pointers	7-11
of strings	7,7-7
precision	7-3,App. A
storage when passed as parameter	App. A
subscripts	7-3,7-4
variable bounds	10,7-3
arrow exponent indicator	4-1,4-4,4-8
arrow separator	4-1,4-2,7-9ff
assignment statement	11,13 to 16,6-1,6-3ff
base of number	34,4-4,4-6
based declarator	4-1,7-1,7-9ff
based variables	7-9 to 7-13
begin	
bracket	4-1,4-3
use in block	1,3-1
use in compound statement	12,6-1
binary literal	33,34,4-6,4-9
bit operation	33,34,4-9
blank space	2,4-1,4-2
blkend run-time routine	App. B
blkstart run-time routine	App. B
block	
begin	1,3-1
definition	1,3-1
inner	8 to 10,2-1
nesting of	8 to 10,2-1
procedure	22,3-1,8-1,8-2
scope of	2-3

Subject	Page
boolean declarator	5,4-1,7-1,7-2
boolean	
expression	19,5-4,6-11
operator	19,4-7
operator precedence	4-8
value	5,4-7
bounds of arrays	7,10,7-3
bracket	4-1,4-3
built-in function	
abs	9-1
address	7-9 to 7-11,9-2
arctan	9-1
cos	9-1
entier	9-2
exp	9-1
index	33,7-7,9-3
length	32,7-7,9-3
ln	9-1
page	29,9-5,10-6
rotate	33,9-4
shift	33,9-4
sign	9-1
sin	9-1
sqrt	9-1
tab	29,31,9-5,10-6
call	
by name	8-5
by value	8-5,8-7
run-time routine	App. B
specification of parameters for	App. A
to function	27,8-1,8-4
to procedure	23,8-1,8-3
channel, I/O	10-1
character string	32,7-6,9-3
close run-time I/O procedure	28,10-3
coding examples	
arithmetic expression	5-2,6-5
array	7-5
assignment statement	6-5,6-6
bit operation	4-10,7-5
boolean expression	6-6
for statement	6-9,6-10
go to statement	6-13
if statement	6-13
pointer expressions	App. D
procedure	8-8
program, sort	App. D
string manipulation	6-10

Subject	Page
colon	2,4-1,4-2,7-13
comma	2,4-1,4-2
comment	12,4-1,4-2
complex declarator	5,4-1,7-1,7-2
complex numbers	4-4,4-5
compount statement	12,6-1
concatenation of string	32,
conditional	
expression	20, Chapter 5,6-12
statement	6-1,6-12
controlled variable	20,6-7
conversion of data types	15,16,5-1,6-3,6-4,12-1
cos built-in function	9-1
data types	7-1,7-2
boolean	5,7-1,7-2
complex	5,7-1,7-2
conversion of	6-3,6-4
definition of	4,7-2
integer	5,7-1,7-2
label	27,7-1,8-8
of function	26,27
pointer	5,7-2,7-9,7-1
real	5,7-1,7-2
string	5,7-1,7-2,7-6
decimal point	4-1,4-2,10-4
declaration	
array	5,10,7-3
based variable or array	7-9
data type	7-2
definition of	1,7-1
external procedure	24,8-3
external variable	7-16
label	9,8-8,7-13
literal	7-17
own variable	4,7-16
pointer	7-2,7-9ff
procedure	22, Chapter 8
string variable or array	5,32,7-8
switch	7-14

Subject	Page
declarators, list of	4-1
delimiter	2,4-1
bracket	4-1,4-3
declarator	4-1,Chapter 7,Chapter 8
operator	4-1
arithmetic	4-4
logical	4-7
relational	4-7
sequential	Chapter 6
specificator	4-1,8-8
transliteration of	4-1
designational expression	5-6
diagnostics	11-1
dimensions of arrays	6,7-3
division	4-1,4-4,4-8
do	4-1,6-7
dope and specifiers	App. A
dummy statement	11,6-1,7-14
e,E	4-1,4-2,4-5
else	12,18,19,4-1,4-3,6-12
end	
bracket	12,13,4-1,4-3
of block	1,3-1,8-1
of compound statement	12,6-1
entier built-in function	6-3,7-4,9-2
equal	4-7
eqv	4-1,4-7,4-8,4-9
error messages	11-1
evaluation of expressions	4-4,4-8
exp built-in function	9-1
exponent	4-2,4-4,4-5,10-5
exponentiation	4-4,4-5,4-8
expression	Chapter 5
arithmetic	5-1 to 5-3
boolean	19,5-4
conditional	20,5-1,5-3,6-2,6-12
designational	5-6
pointer	5-5
simple	14,5-1

Subject	Page
extensions to ALGOL	12-1
external declarator	24,7-16,8-3
false	1-1,4-1,4-7,4-9
for statement	11,20,21,6-1,6-7
formal parameter	25,8-1,8-2,8-5,8-6
formatted output	28 to 31, 10-4 ff
function	
built-in	31, Chapter 9
keywords	1-1
procedure	26,8-1,8-4
referencing	27,8-4 ,8-1
getsp run-time routine	App. B
global identifier	7 to 10, 2-1,3-1
go to statement	11,16 to 18,4-1,6-11
greater than	4-1,4-7,4-8
greater than or equal to	4-1,4-7,4-8
identifier	
declaration of	3 to 10, Chapter 7
definition	1,3,1-1
global	7 to 10, 2-1,3-1
keyword	1,1-1
length	1-1,12-1
local	7 to 10, 2-1,3-1
scope of	Chapter 2, Chapter 3
storage of	3,4,2-2, Chapter 7, App. A
if	
clause	Chapter 5
statement	11,18 to 20,6-12
imp	4-1,4-7,4-8,4-9
index built-in function	33,7-7,9-3
input procedures	28, Chapter 10
integer declarator	5,4-1,7-1,7-2
integer	
multi-precision	7-1,7-2
values	4-5
I/O run-time routines	27 to 31, Chapter 10
jump run-time routines	App. B
jumpc run-time routine	App. B
jumpl run-time routine	App. B
keyword	1,1-1,12-1
label specificator	4-1,8-8

Subject	Page
label	
declaration of	9,2-1,7-13,8-8
definition	17,7-13
designational expression	5-6
in *go to*	6-11
scope of	9,2-1
subscripting	17,6-10,7-14,12-1
length built-in function	32,7-7,9-3
less than	4-1,4-7,4-8
less than or equal	4-1,4-7,4-8
limitations to ALGOL 60	12-1
line of ALGOL source code	2
list	
of expressions in *for*	20,6-7
of identifiers in declaration	7
of output call	10-3
of parameters	8-6
of read call	10-2
of write call	10-2
threaded by pointers	7-13
literal declarator	4-1,7-17
literal	
declarations	4-1,7-17
number of different bases	33,34,4-6,4-9
ln built-in function	9-1
loading ALGOL	App. C
local identifier	7 to 10,2-1,3-1
logical	
expression	5-4
operator	4-7,4-9
value	4-7
loop	6-1,6-7,12
movestr run-time routine	App. B
multiplication	4-1,4-4,4-8
multi-precision integers	7-2,App. A
name, call by	8-5
nesting	
of block	7 to 10,3-1
of subscripts	14
not	4-1,4-7,4-8,4-9
not equal	4-1,4-7,4-8
number	4-5,4-6
octal literal	33,34,4-6,4-9
open I/O routine	27,10-1
operand	
arithmetic	4-4,6-3,5-1
bit	4-9
boolean	19,5-4,6-3
designational	5-6
pointer	5-5
relational	19,5-4

Subject	Page
operator	
arithmetic	4-1,4-4
bit	4-9
logical	4-1,4-7,4-9
precedence of	4-8
relational	4-1,4-7
sequential	4-1, Chapter 6
or	4-1,4-7,4-8
output	
formatted using output call	29 to 31,10-4ff
using write call	28,10-2
overflow checking	7-2,7-3
own declarator	4,2-2,4-1,7-16
page built-in function	29,9-5,10-6
parameter	
actual	25,8-6,8-7,App. A
formal	25,2-2,8-6,8-7
parentheses	2,4-1,4-3,4-8,8-5,8-7
pointer declarator	4-1,7-1,7-2,7-9
pointers	5-5,7-9 to 7-13
power of 10	
in number	4-5
transliteration	4-1,4-5
precedence of operators	4-8
precision	
array	7-3
scalar	3,4,4-5,7-1,7-2
string	3,7-6
in passing parameters	App. A
procedure declarator	22,4-1,7-1,8-1
procedure	
block	22,8-1
body	8-1
call	11,23,6-1,8-3,8-4
call by name	8-5
call by value	8-5
data type of	8-1
declaration	22,7-1,8-1
definition of	22,8-1
external	24,8-3
function	26,8-1,8-2,8-4
identifier	8-2
parameters	25,8-1,8-6,8-7, App. A
recursive (reentrant)	27,8-2
return from	23,8-3,8-4
specificators	4-1,8-8
statement	11,23,6-1,8-3

Subject	Page
program	
basic ALGOL	1,2
writing a	35
quotation marks	4-1,4-3,7-6
radix, changing the	34,4-6
read I/O routine	10-2
real declarator	4,5,4-1,7-1,7-2
real data	4,4-5,7-2
recursive (reentrant) procedure	28,8-2
referencing a function	27,8-1,8-4
relational	
expression	19,5-4
operator	4-1,4-7
value	4-7
return	
from function	8-4
from procedure	23,8-3
run-time routine	App. B
rotate built-in function	33,9-4
run-time routines	
alloc	B-4
array	B-2
blkend	B-3
blkstart	B-3
call	B-1
close	28,10-3
getsp	B-3
I/O	Chapter 10
jump	B-1
jumpc	B-1
jumpl	B-1
movestr	B-4
open	27,10-1
output	29 to 31,10-4ff
read	28,10-2
return	B-1
save	B-1
strcmp	B-4
streq	B-4
string	B-3
subscript	B-3
substr	B-4
write	28,10-2

Subject	Page
save run-time routine	App. B
scope of identifiers	7-10, Chapter 2
semicolon	2,1,4-1,4-2
separator	2,4-1,4-2
sequential operator	4-1, Chapter 6
shape of variables	7-1, App. A
shift built-in function	9-4
sign built-in function	9-1
sin built-in function	9-1
specification of parameters	8-1,8-8
specificator	4-1,8-8
.SP	App. A
sqrt built-in function	9-1
stack frame	App. A
stack pointer	App. A
statement	
assignment	11,13 to 16,6-1,6-3ff
comment	12,4-2
compound	12,6-1
conditional	6-1,6-12
consecutive execution of	6-1
definition of	1,11,6-1
dummy	11,6-1,7-14
for	20,21,6-1,6-7
go to	16 to 18,6-1,6-11
if	18 to 20,6-1,6-12
looping	12,6-1,6-7
procedure	11,6-1,8-3
termination	12
transfer	6-1,6-10
step	21,4-1,4-2,6-7
storage	
allocated (run-time)	App. A
allocation and release	4,8 to 10,2-2,3-1
array	5,7-3, App. A
assigned (compile-time)	App. A
based	7-1, App. A
boolean	3,6-3,7-2, App. A
classes of	7-1, App. A
complex	3,7-1, App. A
external	7-16, App. A
integers	3,7-1,7-2, App. A
local	7-1, App. A
multi-precision integers	7-2, App. A
own	4,2-2,7-16, App. A
pointers	3,6-4,7-2, App. A
procedure	App. A
real	3,4,7-2, App. A
scalars	3,4,7-2, App. A
strings	3,7-6, App. A

Subject	Page
strcmp run-time routine	App. B
streq run-time routine	App. B
string declarator	4,5,32,4-1,7-1,7-2,7-6
string	
arrays and variables	32,7-6
concatenation	32,9-3
nesting of constant -s	7-6
referencing	7-6
run-time routine	App. B
substrings	7-7
subroutine	see procedure
subscript	
in assignment variable	6-3
nesting of	14
of array element	6,7-3,7-4
of controlled variable	6-7
of label or switch	5-6,7-14
run-time routine	App. B
substr run-time routine	App. B
subtraction	4-1,4-4,4-8
switch declarator	4-1,7-15
switches	5-6,6-10,6-12,7-15
symbols	
list of	4-1
transliteration of	4-1
tab built-in function	9-5,10-6
teletypewriter transliterations	4-1,4-4,4-5,4-7
then	18,19,4-1,6-12
transfer of control	
conditional	6-12
to procedure	8-3,8-4
to calling block	8-3,8-4
unconditional	6-1,6-11
transfer function	9-2
true	4-1,4-7
truth table	4-7
types of data	7-2, App. A
unconditional transfer	6-1,6-11
undefined label or switch	6-11
underscore	1-1,12-1
until	21,4-1,4-2,6-7
value specificator	4-1,8-5,8-8

Subject	Page
value	
arithmetic	4-5,5-1,7-2
boolean	4-7,5-4,7-2
call by	8-5,8-7
complex	4-5,7-2
designational	5-6
pointer	5-5,7-2
procedure	8-1,8-4
string	7-2,7-6
use of *own* to retain -	7-17
variable	
controlled	6-7
identifier	3,1-1
in array dimensions	10,7-3
while	21,4-1,4-2,6-7
word	
machine	3
reserved	1-1,12-1
write I/O routine	28,10-2
xor	4-1,4-7,4-9,12-1

Reprinted by the
ASSOCIATION FOR COMPUTING MACHINERY
From *Communications of the ACM* 6 (Jan. 1963), 1-17.

Revised Report on the Algorithmic Language ALGOL 60

PETER NAUR (*Editor*)

J. W. BACKUS, C. KATZ, H. RUTISHAUSER, J. H. WEGSTEIN
F. L. BAUER, J. MCCARTHY, K. SAMELSON, A. VAN WIJNGAARDEN
J. GREEN, A. J. PERLIS, B. VAUQUOIS, M. WOODGER

Dedicated to the Memory of WILLIAM TURANSKI

Reprints distributed by the Association for Computing Machinery, 211 East 43 St., New York 17, N. Y.
Single copies to individuals, no charge; Single copies to companies, 50¢ each;
Multiple copies: first 10, 50¢ each; next 100, 25¢ each; all over 100, 10¢ each.

Revised report on the algorithmic language ALGOL 60

Dedicated to the memory of William Turanski

by

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

Edited by

Peter Naur

The report gives a complete defining description of the international algorithmic language ALGOL 60. This is a language suitable for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers.

The introduction contains an account of the preparatory work leading up to the final conference, where the language was defined. In addition the notions reference language, publication language, and hardware representations are explained.

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

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

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

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

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

The report ends with two detailed examples of the use of the language, and an alphabetic index of definitions.

Contents

	Page
Introduction	2
1. Structure of the language	3
1.1 Formalism for syntactic description	4
2. Basic symbols, identifiers, numbers, and strings. Basic concepts	4
2.1 Letters	4
2.2 Digits. Logical values	4
2.3 Delimiters	4
2.4 Identifiers	5
2.5 Numbers	5
2.6 Strings	5
2.7 Quantities, kinds and scopes	5
2.8 Values and types	6
3. Expressions	6
3.1 Variables	6
3.2 Function designators	6
3.3 Arithmetic expressions	7
3.4 Boolean expressions	8
3.5 Designational expressions	9
4. Statements	9
4.1 Compound statements and blocks	9
4.2 Assignment statements	10
4.3 Go to statements	11
4.4 Dummy statements	11
4.5 Conditional statements	11
4.6 For statements	12
4.7 Procedure statements	12
5. Declarations	14
5.1 Type declarations	14
5.2 Array declarations	14
5.3 Switch declarations	15
5.4 Procedure declarations	15
Examples of procedure declarations	16
Alphabetic index of definitions of concepts and syntactic units	18

Introduction

Background

After the publication*† of a preliminary report on the algorithmic language ALGOL, as prepared at a conference in Zürich in 1958, much interest in the ALGOL language developed.

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper-tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an ALGOL *Bulletin*, edited by Peter Naur, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal and informal ones, were held. These meetings revealed some misunderstandings as to the intent of the group which was primarily responsible for the formulation of the language, but at the same time made it clear that there exists a wide appreciation of the effort involved. As a result of the discussions it was decided to hold an international meeting in January 1960 for improving the ALGOL language and preparing a final report. At a European ALGOL Conference in Paris in November 1959, which was attended by about fifty people, seven European representatives were selected to attend the January 1960 Conference, and they represented the following organizations: Association Française de Calcul, British Computer Society, Gesellschaft für Angewandte Mathematik und Mechanik, and Nederlands Rekenmachine Genootschap. The seven representatives held a final preparatory meeting at Mainz in December 1959.

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

* Preliminary report—International Algebraic Language, *Comm. Assoc. Comp. Mach.*, Vol. 1, No. 12 (1958), p. 8.

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

January 1960 Conference

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

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

April 1962 Conference (Edited by M. Woodger)

A meeting of some of the authors of ALGOL 60 was held on 2–3 April 1962, in Rome, Italy, through the facilities and courtesy of the International Computation Centre. The following were present:

Authors	*Advisers*	*Observer*
F. L. Bauer	M. Paul	W. L. van der Poel (Chairman, IFIP TC 2.1 Working Group ALGOL)
J. Green	R. Franciotti	
C. Katz	P. Z. Ingerman	
R. Kogon (representing J. W. Backus)		
P. Naur		
K. Samelson	G. Seegmüller	
J. H. Wegstein	R. E. Utman	
A. van Wijngaarden		
M. Woodger	P. Landin	

The purpose of the meeting was to correct known errors in, attempt to eliminate apparent ambiguities in, and otherwise clarify the ALGOL 60 Report. Extensions to the language were not considered at the meeting. Various proposals for correction and clarification, that were submitted by interested parties in response to the Questionaire in ALGOL Bulletin No. 14, were used as a guide.

This report constitutes a supplement to the ALGOL 60 Report (*Incorporated with it to form the present revision —Ed.*) which should resolve a number of difficulties therein. Not all of the questions raised concerning the original report could be resolved. Rather than risk hastily drawn conclusions on a number of subtle points, which might create new ambiguities, the committee decided to report only those points which they unanimously felt could be stated in clear and unambiguous fashion.

Questions concerned with the following areas are left for further consideration by Working Group 2.1 of IFIP, in the expectation that current work on advanced programming languages will lead to better resolution:

1. Side effects of functions.
2. The *call by name* concept.

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

3. **Own:** static or dynamic.
4. **For** statement: static or dynamic.
5. Conflict between specification and declaration.

The authors of the ALGOL 60 Report present at the Rome Conference, being aware of the formation of a Working Group on ALGOL by IFIP, accepted that any collective responsibility which they might have with respect to the development, specification and refinement of the ALGOL language will from now on be transferred to that body.

This report has been reviewed by IFIP TC 2 on Programming Languages in August 1962, and has been approved by the Council of the International Federation for Information Processing.

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

Reference Language

1. It is the working language of the committee.
2. It is the defining language.
3. The characters are determined by ease of mutual understanding and not by any computer limitations, coder's notation, or pure mathematical notation.
4. It is the basic reference and guide for compiler builders.
5. It is the guide for all hardware representations.
6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.
7. The main publications of the ALGOL language itself will use the reference representation.

Publication Language

1. The publication language admits variations of the reference language according to usage of printing and handwriting (e.g. subscripts, spaces, exponents, Greek letters).
2. It is used for stating and communicating processes.
3. The characters to be used may be different in different countries, but univocal correspondence with reference representation must be secured.

Hardware Representations

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

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

Reference language	*Publication language*
Subscript brackets [ ]	Lowering of the line between the brackets and removal of the brackets.
Exponentiation ↑	Raising of the exponent.
Parentheses ( )	Any form of parentheses, brackets, braces.
Basis of ten $_{10}$	Raising of the ten and of the following integral number, inserting of the intended multiplication sign.

Description of the Reference Language

Was sich überhaupt sagen lässt, lässt
sich klar sagen; und wovon man nicht
reden kann, darüber muss man schweigen.
Ludwig Wittgenstein.

1. Structure of the language

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

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

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

Statements are supported by declarations which are not themselves computing instructions, but inform the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers, or even the set of rules defining a function. A sequence of declarations followed by a sequence of statements and enclosed between **begin** and **end** constitutes a *block*. Every declaration appears in a block in this way and is valid only for that block.

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

In the sequel the syntax and semantics of the language will be given.*

1.1 Formalism for syntactic description

The syntax will be described with the aid of metalinguistic formulae.† Their interpretation is best explained by an example:

⟨ab⟩ ::= (| [| ⟨ab⟩ (| ⟨ab⟩ ⟨d⟩

Sequences of characters enclosed in the brackets ⟨ ⟩ represent metalinguistic variables whose values are sequences of symbols. The marks ::= and | (the latter with the meaning of *or*) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a recursive rule for the formation of values of the variable ⟨ab⟩. It indicates that ⟨ab⟩ may have the value (or [or that given some legitimate value of ⟨ab⟩, another may be formed by following it with the character (or by following it with some value of the variable ⟨d⟩. If the values of ⟨d⟩ are the decimal digits, some values of ⟨ab⟩ are:

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

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables (i.e. the sequences of characters appearing within the brackets ⟨ ⟩ as ab in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In addition some formulae have been given in more than one place.

Definition:
⟨empty⟩ ::=
(i.e. the null string of symbols).

2. Basic symbols, identifiers, numbers, and strings
Basic concepts

The reference language is built up from the following basic symbols:
⟨basic symbol⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨logical value⟩ | ⟨delimiter⟩

2.1 Letters

⟨letter⟩ ::= *a*|*b*|*c*|*d*|*e*|*f*|*g*|*h*|*i*|*j*|*k*|*l*|*m*|*n*|*o*|*p*|*q*|*r*|*s*|*t*|*u*|*v*|*w*|*x*|*y*|*z*|
A|*B*|*C*|*D*|*E*|*F*|*G*|*H*|*I*|*J*|*K*|*L*|*M*|*N*|*O*|*P*|*Q*|*R*|*S*|*T*|*U*|*V*|*W*|*X*|*Y*|*Z*

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

Letters do not have individual meaning. They are used for forming identifiers and strings* (cf. sections 2.4 Identifiers, 2.6 Strings).

2.2.1 *Digits*

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

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

2.2.2 *Logical values*

⟨logical value⟩ ::= **true** | **false**

The logical values have a fixed obvious meaning.

2.3 Delimiters

⟨delimiter⟩ ::= ⟨operator⟩ | ⟨separator⟩ | ⟨bracket⟩ | ⟨declarator⟩ | ⟨specificator⟩
⟨operator⟩ ::= ⟨arithmetic operator⟩ | ⟨relational operator⟩ | ⟨logical operator⟩ | ⟨sequential operator⟩
⟨arithmetic operator⟩ ::= $+ \mid - \mid \times \mid / \mid \div \mid \uparrow$
⟨relational operator⟩ ::= $< \mid \leqslant \mid = \mid \geqslant \mid > \mid \neq$
⟨logical operator⟩ ::= $\equiv \mid \supset \mid \vee \mid \wedge \mid \neg$
⟨sequential operator⟩ ::= **go to** | **if** | **then** | **else** | **for** | **do**†
⟨separator⟩ ::= , | . | $_{10}$ | : | ; | := | ␣ | **step** | **until** | **while** | **comment**
⟨bracket⟩ ::= (|) | [|] | ‘ | ’ | **begin** | **end**

* Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is left undefined or said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the execution of the computation.

† Cf. J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference," ICIP, Paris, June 1959.

* It should be particularly noted that throughout the reference language underlining (*for typographical reasons* **bold type** *is used synonymously—Ed.*) is used for defining independent basic symbols (see sections 2.2.2 and 2.3). These are understood to have no relation to the individual letters of which they are composed. Within the present report underlining will be used for no other purpose.

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

⟨declarator⟩ ::= **own** | **Boolean** | **integer** | **real** | **array** | **switch** | **procedure**
⟨specificator⟩ ::= **string** | **label** | **value**

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

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

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

The sequence of basic symbols:	is equivalent to
; **comment** ⟨any sequence not containing; ⟩ ;	;
begin comment ⟨any sequence not containing ; ⟩ ;	**begin**
end ⟨any sequence not containing **end** or ; or **else**⟩	**end**.

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

2.4 Identifiers

2.4.1 *Syntax*

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩ | ⟨identifier⟩ ⟨digit⟩

2.4.2 *Examples*

q
Soup
V17a
a34kTMNs
MARILYN

2.4.3 *Semantics*

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

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf. section 2.7 Quantities, kinds and scopes, and section 5 Declarations).

2.5 Numbers

2.5.1 *Syntax*

⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨unsigned integer⟩ ⟨digit⟩
⟨integer⟩ ::= ⟨unsigned integer⟩ | + ⟨unsigned integer⟩ | − ⟨unsigned integer⟩
⟨decimal fraction⟩ ::= . ⟨unsigned integer⟩
⟨exponent part⟩ ::= $_{10}$ ⟨integer⟩
⟨decimal number⟩ ::= ⟨unsigned integer⟩ | ⟨decimal fraction⟩ | ⟨unsigned integer⟩ ⟨decimal fraction⟩
⟨unsigned number⟩ ::= ⟨decimal number⟩ | ⟨exponent part⟩ | ⟨decimal number⟩ ⟨exponent part⟩
⟨number⟩ ::= ⟨unsigned number⟩ | + ⟨unsigned number⟩ | − ⟨unsigned number⟩

2.5.2 *Examples*

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

2.5.3 *Semantics*

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

2.5.4 *Types*

Integers are of type **integer.** All other numbers are of type **real** (cf. section 5.1 Type declarations).

2.6 Strings

2.6.1 *Syntax*

⟨proper string⟩ ::= ⟨any sequence of basic symbols not containing ‘ or ’ ⟩ | ⟨empty⟩
⟨open string⟩ ::= ⟨proper string⟩ | ‘⟨open string⟩’ | ⟨open string⟩ ⟨open string⟩
⟨string⟩ ::= ‘⟨open string⟩’

2.6.2 *Examples*

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

2.6.3 *Semantics*

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

Strings are used as actual parameters of procedures (c.f. sections 3.2 Function designators and 4.7 Procedure statements).

2.7 Quantities, kinds and scopes

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

The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see section 4.1.3.

2.8 Values and types

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

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

The various *types* (**integer, real, Boolean**) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are *arithmetic*, *Boolean*, and *designational*, expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential, operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

⟨expression⟩ :: = ⟨arithmetic expression⟩ | ⟨Boolean expression⟩ | ⟨designational expression⟩

3.1 Variables

3.1.1 *Syntax*

⟨variable identifier⟩ :: = ⟨identifier⟩
⟨simple variable⟩ :: = ⟨variable identifier⟩
⟨subscript expression⟩ :: = ⟨arithmetic expression⟩
⟨subscript list⟩ :: = ⟨subscript expression⟩ |
    ⟨subscript list⟩, ⟨subscript expression⟩
⟨array identifier⟩ :: = ⟨identifier⟩
⟨subscripted variable⟩ :: =
    ⟨array identifier⟩ [⟨subscript list⟩]
⟨variable⟩ :: = ⟨simple variable⟩ |
    ⟨subscripted variable⟩

3.1.2 *Examples*
epsilon
detA
*a*17
Q[7, 2]
x[*sin*(*n*×*pi*/2), *Q*[3, *n*, 4]]

3.1.3 *Semantics*

A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2). The type of the value of a particular variable is defined in the declaration for the variable itself (cf. section 5.1 Type declarations) or for the corresponding array identifier (cf. section 5.2 Array declarations).

3.1.4 *Subscripts*

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

3.1.4.2 Each subscript position acts like a variable of type **integer** and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (cf. section 4.2.4). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. section 5.2 Array declarations).

3.2 Function designators

3.2.1 *Syntax*

⟨procedure identifier⟩ :: = ⟨identifier⟩
⟨actual parameter⟩ :: = ⟨string⟩ | ⟨expression⟩ |
    ⟨array identifier⟩ | ⟨switch identifier⟩ |
    ⟨procedure identifier⟩
⟨letter string⟩ :: = ⟨letter⟩ | ⟨letter string⟩ ⟨letter⟩
⟨parameter delimiter⟩ :: = , |) ⟨letter string⟩ : (
⟨actual parameter list⟩ :: = ⟨actual parameter⟩ |
    ⟨actual parameter list⟩ ⟨parameter delimiter⟩
    ⟨actual parameter⟩
⟨actual parameter part⟩ :: = ⟨empty⟩ |
    (⟨actual parameter list⟩)
⟨function designator⟩ :: = ⟨procedure identifier⟩
    ⟨actual parameter part⟩

3.2.2 *Examples*

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

3.2.3 *Semantics*

Function designators define single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration (cf. section 5.4 Procedure declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7 Procedure statements. Not every procedure declaration defines the value of a function designator.

3.2.4 *Standard functions*

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

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

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

3.2.5 *Transfer functions*

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

entier(E) ,

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

3.3 Arithmetic expressions

3.3.1 *Syntax*

⟨adding operator⟩ ::= + | −
⟨multiplying operator⟩ ::= × | / | ÷
⟨primary⟩ ::= ⟨unsigned number⟩ | ⟨variable⟩ | ⟨function designator⟩ | (⟨arithmetic expression⟩)
⟨factor⟩ ::= ⟨primary⟩ | ⟨factor⟩ ↑ ⟨primary⟩
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ ⟨multiplying operator⟩ ⟨factor⟩
⟨simple arithmetic expression⟩ ::= ⟨term⟩ | ⟨adding operator⟩ ⟨term⟩ | ⟨simple arithmetic expression⟩ ⟨adding operator⟩ ⟨term⟩
⟨if clause⟩ ::= **if** ⟨Boolean expression⟩ **then**
⟨arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩ | ⟨if clause⟩ ⟨simple arithmetic expression⟩ **else** ⟨arithmetic expression⟩

3.3.2 *Examples*

Primaries:
$7.394_{10}{-8}$
sum
$w[i + 2, 8]$
$cos\,(y + z \times 3)$
$(a - 3/y + vu \uparrow 8)$

Factors:
omega
$sum \uparrow cos\,(y + z \times 3)$
$7.394_{10} - 8 \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu \uparrow 8)$

Terms:
U
$omega \times sum \uparrow cos\,(y + z \times 3)/7.394_{10} - 8 \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu \uparrow 8)$

Simple arithmetic expression:
$U - Yu + omega \times sum \uparrow cos\,(y + z \times 3)/7.394_{10}{-8} \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu \uparrow 8)$

Arithmetic expressions:
$w \times u - Q(S + Cu) \uparrow 2$
if $q > 0$ **then** $S + 3 \times Q/A$ **else** $2 \times S + 3 \times q$
if $a < 0$ **then** $U + V$ **else if** $a \times b > 17$ **then** U/V **else if** $k \neq y$ **then** V/U **else** 0
$a \times sin\,(omega \times t)$
$0.57_{10}12 \times a\,[N \times (N - 1)/2, 0]$
$(A \times arctan(y) + Z) \uparrow (7 + Q)$
if q **then** $n - 1$ **else** n
if $a < 0$ **then** A/B **else if** $b = 0$ **then** B/A **else** z

3.3.3 *Semantics*

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure (cf. section 5.4.4 Values of function designators) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

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

else ⟨simple arithmetic expression⟩

is equivalent to the construction:

else if true then ⟨simple arithmetic expression⟩

3.3.4 *Operators and types*

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

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

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

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

means

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

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

$$a \div b = sign\ (a/b) \times entier\ (abs(a/b))$$

(cf. sections 3.2.4 and 3.2.5).

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

$$2 \uparrow n \uparrow k \quad \text{means } (2^n)^k$$

while

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

Writing i for a number of **integer** type, r for a number of **real** type, and a for a number of either **integer** or **real** type, the result is given by the following rules:

$a \uparrow i$ — If $i > 0$, $a \times a \times \ldots \times a$ (i times), of the same type as a.
If $i = 0$, if $a \neq 0$, 1, of the same type as a, if $a = 0$, undefined.
If $i < 0$, if $a \neq 0$, $1/(a \times a \times \ldots \times a)$ (the denominator has $-i$ factors), of type **real**, if $a = 0$, undefined.

$a \uparrow r$ — If $a > 0$, $exp\ (r \times ln(a))$, of type **real**.
If $a = 0$, if $r > 0$, 0 . 0, of type **real**, if $r < 0$, undefined.
If $a < 0$, always undefined.

3.3.5 *Precedence of operators*

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

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

first: ↑
second: $\times$ / $\div$
third: $+$ $-$

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

3.3.6 *Arithmetics of* **real** *quantities*

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

3.4 Boolean expressions

3.4.1 *Syntax*

⟨relational operator⟩ ::= $<$ | $\leqslant$ | $=$ | $\geqslant$ | $>$ | $\neq$
⟨relation⟩ ::=
 ⟨simple arithmetic expression⟩ ⟨relational operator⟩ ⟨simple arithmetic expression⟩
⟨Boolean primary⟩ ::= ⟨logical value⟩ | ⟨variable⟩ |
 ⟨function designator⟩ | ⟨relation⟩ |
 (⟨Boolean expression⟩)
⟨Boolean secondary⟩ ::= ⟨Boolean primary⟩ |
 ¬ ⟨Boolean primary⟩
⟨Boolean factor⟩ ::= ⟨Boolean secondary⟩ |
 ⟨Boolean factor⟩ ∧ ⟨Boolean secondary⟩
⟨Boolean term⟩ ::= ⟨Boolean factor⟩ |
 ⟨Boolean term⟩ ∨ ⟨Boolean factor⟩
⟨implication⟩ ::= ⟨Boolean term⟩ |
 ⟨implication⟩ ⊃ ⟨Boolean term⟩
⟨simple Boolean⟩ ::= ⟨implication⟩ |
 ⟨simple Boolean⟩ ≡ ⟨implication⟩
⟨Boolean expression⟩ ::= ⟨simple Boolean⟩ | ⟨if clause⟩
 ⟨simple Boolean⟩ **else** ⟨Boolean expression⟩

3.4.2 *Examples*

$x = -2$
$Y > V \vee z < q$
$a + b > -5 \wedge z - d > q \uparrow 2$
$p \wedge q \vee x \neq y$
$g \equiv \neg a \wedge b \wedge \neg c \vee d \vee e \supset \neg f$
if $k < 1$ **then** $s > w$ **else** $h < c$
if if if a **then** b **else** c **then** d **else** f **then** g **else** $h < k$

3.4.3 *Semantics*

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

3.4.4 *Types*

Variables and function designators entered as Boolean primaries must be declared **Boolean** (cf. section 5.1 Type declarations and section 5.4.4 Values of function designators).

3.4.5 *The operators*

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

The meaning of the logical operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\supset$ (implies), and $\equiv$ (equivalent), is given by the following function table.

$b1$	**false**	**false**	**true**	**true**
$b2$	**false**	**true**	**false**	**true**
$\neg\ b1$	**true**	**true**	**false**	**false**
$b1 \wedge b2$	**false**	**false**	**false**	**true**
$b1 \vee b2$	**false**	**true**	**true**	**true**
$b1 \supset b2$	**true**	**true**	**false**	**true**
$b1 \equiv b2$	**true**	**false**	**false**	**true**

3.4.6 *Precedence of operators*

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

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

first: arithmetic expressions according to section 3.3.5.
second: $< \quad \leqslant \quad = \quad \geqslant \quad > \quad \neq$
third: $\neg$
fourth: $\wedge$
fifth: $\vee$
sixth: $\supset$
seventh: $\equiv$

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

3.5 Designational expressions

3.5.1 *Syntax*

⟨label⟩ :: = ⟨identifier⟩ | ⟨unsigned integer⟩
⟨switch identifier⟩ :: = ⟨identifier⟩
⟨switch designator⟩ :: =
 ⟨switch identifier⟩[⟨subscript expression⟩]
⟨simple designational expression⟩ :: = ⟨label⟩ |
 ⟨switch designator⟩ | (⟨designational expression⟩)
⟨designational expression⟩ :: = ⟨simple designational expression⟩ | ⟨if clause⟩ ⟨simple designational expression⟩ **else** ⟨designational expression⟩

3.5.2 *Examples*

17
*p*9
Choose [$n - 1$]
Town [**if** $y < 0$ **then** N **else** $N + 1$]
if $Ab < c$ **then** 17 **else** q [**if** $w \leqslant 0$ **then** 2 **else** n]

3.5.3 *Semantics*

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

3.5.4 *The subscript expression*

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

3.5.5 *Unsigned integers as labels*

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

4. Statements

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

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

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

4.1 Compound statements and blocks

4.1.1 *Syntax*

⟨unlabelled basic statement⟩ :: =
 ⟨assignment statement⟩ | ⟨go to statement⟩ |
 ⟨dummy statement⟩ | ⟨procedure statement⟩
⟨basic statement⟩ :: = ⟨unlabelled basic statement⟩ |
 ⟨label⟩ : ⟨basic statement⟩
⟨unconditional statement⟩ :: = ⟨basic statement⟩ |
 ⟨compound statement⟩ | ⟨block⟩
⟨statement⟩ :: = ⟨unconditional statement⟩ |
 ⟨conditional statement⟩ | ⟨for statement⟩
⟨compound tail⟩ :: = ⟨statement⟩ **end** |
 ⟨statement⟩ ; ⟨compound tail⟩

⟨block head⟩ :: = **begin** ⟨declaration⟩ |
  ⟨block head⟩ ; ⟨declaration⟩
⟨unlabelled compound⟩ :: = **begin** ⟨compound tail⟩
⟨unlabelled block⟩ :: =
  ⟨block head⟩ ; ⟨compound tail⟩
⟨compound statement⟩ :: = ⟨unlabelled compound⟩ |
  ⟨label⟩ : ⟨compound statement⟩
⟨block⟩ :: = ⟨unlabelled block⟩ | ⟨label⟩ : ⟨block⟩
⟨program⟩ :: = ⟨block⟩ | ⟨compound statement⟩

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

Compound statement:
 L: L: . . . **begin** S ; S ; . . . S ; S **end**

Block:
 L: L: . . . **begin** D ; D ; . . . D ; S ; S ; . . . S ; S **end**

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

4.1.2 *Examples*

Basic statements:
 $a := p + q$
 go to *Naples*
 START: *CONTINUE*: $W := 7.993$

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

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

4.1.3 *Semantics*

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

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

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

4.2 Assignment statements

4.2.1 *Syntax*

⟨left part⟩ :: = ⟨variable⟩ := |
  ⟨procedure identifier⟩ :=
⟨left part list⟩ ::= ⟨left part⟩ |
  ⟨left part list⟩ ⟨left part⟩
⟨assignment statement⟩ :: = ⟨left part list⟩ ⟨arithmetic expression⟩ | ⟨left part list⟩ ⟨Boolean expression⟩

4.2.2 *Examples*

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

4.2.3 *Semantics*

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

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

4.2.3.2 The expression of the statement is evaluated.

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

4.2.4 *Types*

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

$$entier\ (E + 0.5)$$

where E is the value of the expression. The type asso-

ciated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (cf. section 5.4.4).

4.3 Go to statements

4.3.1 *Syntax*

⟨go to statement⟩ :: = **go to** ⟨designational expression⟩

4.3.2 *Examples*

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

4.3.3 *Semantics*

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

4.3.4 *Restriction*

Since labels are inherently local, no **go to** statement can lead from outside into a block. A **go to** statement may, however, lead from outside into a compound statement.

4.3.5 **Go to** *an undefined switch designator*

A **go to** statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined.

4.4 Dummy statements

4.4.1 *Syntax*

⟨dummy statement⟩ :: = ⟨empty⟩

4.4.2 *Examples*

L:
begin . . . ; *John*: **end**

4.4.3 *Semantics*

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

4.5 Conditional statements

4.5.1 *Syntax*

⟨if clause⟩ :: = **if** ⟨Boolean expression⟩ **then**
⟨unconditional statement⟩ :: = ⟨basic statement⟩ |
⟨compound statement⟩ | ⟨block⟩
⟨if statement⟩ :: =
⟨if clause⟩ ⟨unconditional statement⟩
⟨conditional statement⟩ :: = ⟨if statement⟩ |
⟨if statement⟩ **else** ⟨statement⟩ |
⟨if clause⟩ ⟨for statement⟩ |
⟨label⟩ : ⟨conditional statement⟩

4.5.2 *Examples*

if $x > 0$ **then** $n := n + 1$
if $v > u$ **then** V: $q := n + m$ **else go to** R
if $s < 0 \vee P < Q$ **then** AA: **begin if** $q < v$ **then** $a := v/s$
else $y := 2 \times a$ **end**
else if $v > s$ **then** $a := v - q$
else if $v > s - 1$ **then go to** S

4.5.3 *Semantics*

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

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

4.5.3.2 *Conditional statement*. According to the syntax two different forms of conditional statements are possible. These may be illustrated as follows:

if B1 **then** S1 **else if** B2 **then** S2 **else** S3; S4

and

if B1 **then** S1 **else if** B2 **then** S2 **else if** B3 **then** S3 ; S4

Here B1 to B3 are Boolean expressions, while S1 to S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows: The Boolean expressions of the **if** clauses are evaluated one after the other in sequence from left to right until one yielding the value **true** is found. Then the unconditional statement following this Boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be S4, the statement following the complete conditional statement. Thus the effect of the delimiter **else** may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction
else ⟨unconditional statement⟩
is equivalent to
else if true then ⟨unconditional statement⟩

If none of the Boolean expressions of the **if** clauses is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation the following picture may be useful:

```
              ...................................
              ↑                     ↑          ↓
if B1 then S1 else if B2 then S2 else S3 ; S4
   ↓                   ↑  ↓                 ↑
   .....................  ...................
         B1 false              B2 false
```

4.5.4 **Go to** *into a conditional statement*

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

4.6 For statements

4.6.1 *Syntax*

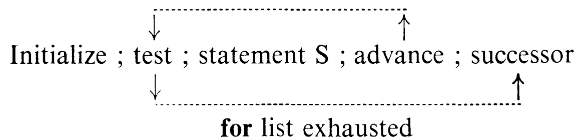
⟨for list element⟩ : : = ⟨arithmetic expression⟩ |
  ⟨arithmetic expression⟩ **step** ⟨arithmetic expression⟩
  until ⟨arithmetic expression⟩ |
  ⟨arithmetic expression⟩ **while** ⟨Boolean expression⟩
⟨for list⟩ : : = ⟨for list element⟩ |
  ⟨for list⟩ , ⟨for list element⟩
⟨for clause⟩ : : = **for** ⟨variable⟩ := ⟨for list⟩ **do**
⟨for statement⟩ : : = ⟨for clause⟩ ⟨statement⟩ |
  ⟨label⟩ : ⟨for statement⟩

4.6.2 *Examples*

for q := 1 **step** s **until** n **do** $A[q] := B[q]$
for k := 1, $V1 \times 2$ **while** $V1 < N$ **do**
  for $j := I + G, L,$ 1 **step** 1 **until** $N, C + D$ **do**
  $A[k, j] := B[k, j]$

4.6.3 *Semantics*

A **for** clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable. The process may be visualized by means of the following picture:



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

4.6.4 *The* **for** *list elements*

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

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

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

  $V := A$;
$L1$: **if** $(V - C) \times sign(B) > 0$ **then go to** *Element exhausted*;
  Statement S;
  $V := V + B$;
  go to $L1$;

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

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

  $L3$: $V := E$;
  if $\neg$ F **then go to** *Element exhausted*;
  Statement S;
  go to $L3$;

where the notation is the same as in 4.6.4.2 above.

4.6.5 *The value of the controlled variable upon exit*

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

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

4.6.6 **Go to** *leading into a* **for** *statement*

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

4.7 Procedure statements

4.7.1 *Syntax*

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

4.7.2 *Examples*

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

These examples correspond to examples given in section 5.4.2.

4.7.3 *Semantics*

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

4.7.3.1. *Value assignment* (*call by value*). All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8 Values and types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. section 5.4.5). As a consequence, variables called by value are to be considered as non-local to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).

4.7.3.2 *Name replacement* (*call by name*). Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3 *Body replacement and execution.* Finally the procedure body, modified as above, is inserted in place of the procedure statement, and executed. If the procedure is called from a place outside the scope of any non-local quantity of the procedure body, the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

4.7.4 *Actual-formal correspondence*

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

4.7.5 *Restrictions*

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

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

4.7.5.1 If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL 60 statement (as opposed to non-ALGOL code, cf. section 4.7.8), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL code.

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

4.7.5.3 A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition, if the formal parameter is called by value, the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4 A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier or a string, because these latter do not possess values (the exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (cf. section 5.4.1) and which defines the value of a function designator (cf. section 5.4.4). This procedure identifier is in itself a complete expression).

4.7.5.5 Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement such restrictions must evidently be observed.

4.7.6 Deleted.

4.7.7 *Parameter delimiters*

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

4.7.8 *Procedure body expressed in code*

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

5. Declarations

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. section 4.1.3).

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

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

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

Syntax

⟨declaration⟩ :: = ⟨type declaration⟩ |
⟨array declaration⟩ |
⟨switch declaration⟩ |
⟨procedure declaration⟩

5.1 Type declarations

5.1.1 *Syntax*

⟨type list⟩ :: = ⟨simple variable⟩ |
⟨simple variable⟩, ⟨type list⟩
⟨type⟩ :: = **real** | **integer** | **Boolean**
⟨local or own type⟩ :: = ⟨type⟩ | **own** ⟨type⟩
⟨type declaration⟩ :: = ⟨local or own type⟩ ⟨type list⟩

5.1.2 *Examples*

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

5.1.3 *Semantics*

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

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

For the semantics of **own,** see the fourth paragraph of section 5 above.

5.2 Array declarations

5.2.1 *Syntax*

⟨lower bound⟩ :: = ⟨arithmetic expression⟩
⟨upper bound⟩ :: = ⟨arithmetic expression⟩
⟨bound pair⟩ :: = ⟨lower bound⟩ : ⟨upper bound⟩
⟨bound pair list⟩ :: = ⟨bound pair⟩ |
⟨bound pair list⟩, ⟨bound pair⟩
⟨array segment⟩:: =
⟨array identifier⟩ [⟨bound pair list⟩] |
⟨array identifier⟩, ⟨array segment⟩
⟨array list⟩ :: = ⟨array segment⟩ | ⟨array list⟩ , ⟨array segment⟩
⟨array declaration⟩ :: = **array** ⟨array list⟩ |
⟨local or own type⟩ **array** ⟨array list⟩

5.2.2 *Examples*

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

5.2.3 *Semantics*

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

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

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

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

5.2.4 *Lower upper bound expressions*

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

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

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

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

5.2.5 *The identity of subscripted variables*

The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. However, even if an array is declared **own** the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3 Switch declarations

5.3.1 *Syntax*

⟨switch list⟩ :: = ⟨designational expression⟩ |
    ⟨switch list⟩, ⟨designational expression⟩
⟨switch declaration⟩ :: =
    switch ⟨switch identifier⟩ := ⟨switch list⟩

5.3.2 *Examples*

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

5.3.3 *Semantics*

A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1, 2, . . . , obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (cf. section 3.5 Designational expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4 *Evaluation of expressions in the switch list*

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

5.3.5 *Influence of scopes*

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

5.4 Procedure declarations

5.4.1 *Syntax*

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

5.4.2 *Examples* (see also the examples at the end of the report)

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

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

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

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

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

5.4.3 *Semantics*

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (cf. section 3.2 Function designators and section 4.7 Procedure statements) be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or non-local to the body depending on whether they are declared within the body or not. Those of them which are non-local to the body may well be local to the block in the head of which the procedure declaration appears. The procedure body always acts like a block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

5.4.4 *Values of function designators*

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

5.4.5 *Specifications*

In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part no formal parameter may occur more than once. Specifications of formal parameters called by value (cf. section 4.7.3.1) must be supplied and specifications of formal parameters called by name (cf. section 4.7.3.2) may be omitted.

5.4.6 *Code as procedure body*

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

Examples of procedure declarations

Example 1

```
procedure euler (fct, sum, eps, tim) ; value eps, tim ;
  integer tim ;
real procedure fct ; real sum, eps ;
comment euler computes the sum of fct(i) for i from zero
up to infinity by means of a suitably refined euler trans-
formation. The summation is stopped as soon as tim
times in succession the absolute value of the terms of the
transformed series are found to be less than eps. Hence,
one should provide a function fct with one integer argument,
an upper bound eps, and an integer tim. The output is the
sum sum. euler is particularly efficient in the case of a
slowly convergent or divergent alternating series ;
begin integer i, k, n, t ; array m[0 : 15] ; real mn, mp, ds ;
  i := n := t := 0 ; m[0] := fct(0) ; sum := m[0]/2 ;
nextterm: i := i + 1 ; mn := fct(i) ;
          for k := 0 step 1 until n do
            begin mp := (mn + m[k])/2; m[k] := mn ;
              mn := mp end means ;
          if (abs(mn) < abs(m[n])) ∧ (n < 15) then
            begin ds := mn/2 ; n := n + 1 ;
              m[n] := mn end accept
          else ds := mn ;
          sum := sum + ds ;
          if abs(ds) < eps then t := t + 1 else t := 0 ;
          if t < tim then go to nextterm
end euler
```

Example 2*

```
procedure RK(x, y, n, FKT, eps, eta, xE, yE, fi) ;
  value x, y ; integer n ;
Boolean fi ; real x, eps, eta, xE ; array y, yE ;
  procedure FKT ;
comment : RK integrates the system
        y'_k = f_k(x, y_1, y_2, . . ., y_n)   (k = 1, 2, . . . n)
of differential equations with the method of Runge-Kutta
with automatic search for appropriate length of inte-
gration step. Parameters are: The initial values x and
y[k] for x and the unknown functions y_k(x). The order n
of the system. The procedure FKT(x, y, n, z) which
represents the system to be integrated, i.e. the set of
functions f_k. The tolerance values eps and eta which
govern the accuracy of the numerical integration. The
end of the integration interval xE. The output parameter
yE which represents the solution at x = xE. The Boolean
variable fi, which must always be given the value true
for an isolated or first entry into RK. If, however, the
functions y must be available at several meshpoints
x_0, x_1, . . ., x_n, then the procedure must be called repeatedly
(with x = x_k, xE = x_{k+1}, for k = 0, 1, . . ., n - 1) and
then the later calls may occur with fi = false which saves
computing time. The input parameters of FKT must be
x, y, n, the output parameter z represents the set of
derivatives z[k] = f_k(x, y[1], y[2], . . ., y[n]) for x and the
actual y's. A procedure comp enters as a non-local
identifier ;

begin
  array z, y1, y2, y3[1 : n] ; real x1, x2, x3, H ;
    Boolean out ;
  integer k, j ; own real s, Hs ;
  procedure RK1ST(x, y, h, xe, ye) ; real x, h, xe ;
      array y, ye ;
    comment : RK1ST integrates one single RUNGE-
    KUTTA step with initial values x, y[k] which yields
    the output parameters xe = x + h and ye[k], the
    latter being the solution at xe. IMPORTANT: the
    parameters n, FKT, z enter RK1ST as non-local
    entities ;
    begin
      array w[1 : n], a[1 : 5] ; integer k, j ;
      a[1] := a[2] := a[5] := h/2 ; a[3] := a[4] := h ;
      xe := x ;
      for k := 1 step 1 until n do ye[k] := w[k] := y[k] ;
      for j := 1 step 1 until 4 do
      begin
        FKT(xe, w, n, z) ;
        xe := x + a[j] ;
        for k := 1 step 1 until n do
        begin
          w[k] := y[k] + a[j] × z[k] ;
          ye[k] := ye[k] + a[j + 1] × z[k]/3
        end k
      end j
    end RK1ST ;

BEGIN OF PROGRAM:
  if fi then begin H := xE - x; s := 0 end
  else H := Hs ; out := false ;

AA: if (x + 2.01 × H - xE > 0) ≡ (H > 0) then
    begin Hs := H ; out := true ; H := (xE - x)/2
    end if ;
    RK1ST(x, y, 2 × H, x1, y1) ;
BB: RK1ST(x, y, H, x2, y2) ; RK1ST(x2, y2, H, x3, y3);
    for k := 1 step 1 until n do
        if comp(y1[k], y3[k], eta) > eps then go to CC ;
    comment : comp(a, b, c) is a function designator, the
    value of which is the absolute value of the difference
    of the mantissae of a and b, after the exponents of
    these quantities have been made equal to the largest
    of the exponents of the originally given parameters
    a, b, c ;
    x := x3 ; if out then go to DD ;
    for k := 1 step 1 until n do y[k] := y3[k] ;
    if s = 5 then begin s := 0 ; H := 2 × H end if ;
    s := s + 1 ; go to AA ;
CC: H := 0.5 × H ; out := false ; x1 := x2 ;
    for k := 1 step 1 until n do y1[k] := y2[k] ;
    go to BB ;
DD: for k := 1 step 1 until n do yE[k] := y3[k]
end RK
```

* This RK-program contains some new ideas which are related to ideas of S. Gill, "A process for the step by step integration of differential equations in an automatic computing machine," *Proc. Camb. Phil. Soc.*, Vol. 47 (1951), p. 96, and C. E Fröberg, "On the solution of ordinary differential equations with digital computing machines," *Fysiograf Sällsk. Lund*, Förhd 20, Nr. 11 (1950), pp. 136–52. It must be clear, however, that with respect to computing time and round-off errors it may not be optimal, nor has it actually been tested on a computer.

For alphabetic index, see next page

Alphabetic index of definitions of concepts and syntactic units

All references are given through section numbers. The references are given in three groups:

def Following the abbreviation "def" reference to the syntactic definition (if any) is given.

synt Following the abbreviation "synt" references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.

text Following the word "text" the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined* words have been collected at the beginning. The examples have been ignored in compiling the index.

* Bold faced.—*Ed.*

+, see: plus
−, see: minus
×, see: multiply
/, ÷, see: divide
↑, see: exponentiation
<, ⩽, =, ⩾, >, ≠, see: ⟨relational operator⟩
≡, ⊃, ∨, ∧, ¬, see: ⟨logical operator⟩
,, see: comma
., see: decimal point
$_{10}$, see: ten
:, see: colon
;, see: semicolon
:=, see: colon equal
␣, see: space
( ), see: parentheses
[ ], see: subscript bracket
⌜ ⌝, see: string quote
⟨actual parameter⟩, def 3.2.1, 4.7.1
⟨actual parameter list⟩, def 3.2.1, 4.7.1
⟨actual parameter part⟩, def 3.2.1, 4.7.1
⟨adding operator⟩, def 3.3.1
alphabet, text 2.1
arithmetic, text 3.3.6
⟨arithmetic expression⟩, def 3.3.1 synt 3, 3.1.1, 3.3.1, 3.4.1, 4.2.1, 4.6.1, 5.2.1 text 3.3.3
⟨arithmetic operator⟩, def 2.3 text 3.3.4
array, synt 2.3, 5.2.1, 5.4.1
array, text 3.1.4.1
⟨array declaration⟩, def 5.2.1 synt 5 text 5.2.3
⟨array identifier⟩, def 3.1.1 synt 3.2.1, 4.7.1, 5.2.1 text 2.8
⟨array list⟩, def 5.2.1
⟨array segment⟩, def 5.2.1
⟨assignment statement⟩, def 4.2.1 synt 4.1.1 text 1, 4.2.3
⟨basic statement⟩, def 4.1.1 synt 4.5.1
⟨basic symbol⟩, def 2
begin, synt 2.3, 4.1.1
⟨block⟩, def 4.1.1 synt 4.5.1 text 1, 4.1.3, 5
⟨block head⟩, def 4.1.1
Boolean, synt 2.3, 5.1.1 text 5.1.3
⟨Boolean expression⟩, def 3.4.1 synt 3, 3.3.1, 4.2.1, 4.5.1, 4.6.1 text 3.4.3
⟨Boolean factor⟩, def 3.4.1
⟨Boolean primary⟩, def 3.4.1
⟨Boolean secondary⟩, def 3.4.1
⟨Boolean term⟩, def 3.4.1
⟨bound pair⟩, def 5.2.1
⟨bound pair list⟩, def 5.2.1
⟨bracket⟩, def 2.3
⟨code⟩, synt 5.4.1 text 4.7.8, 5.4.6
colon :, synt 2.3, 3.2.1, 4.1.1, 4.5.1, 4.6.1, 4.7.1, 5.2.1
colon equal :=, synt 2.3, 4.2.1, 4.6.1, 5.3.1
comma , , synt 2.3, 3.1.1, 3.2.1, 4.6.1, 4.7.1, 5.1.1, 5.2.1, 5.3.1, 5.4.1
comment, synt 2.3
comment convention, text 2.3
⟨compound statement⟩, def 4.1.1 synt 4.5.1 text 1
⟨compound tail⟩, def 4.1.1
⟨conditional statement⟩, def 4.5.1 synt 4.1.1 text 4.5.3
⟨decimal fraction⟩, def 2.5.1
⟨decimal number⟩, def 2.5.1 text 2.5.3
decimal point ., synt 2.3, 2.5.1
⟨declaration⟩, def 5 synt 4.1.1 text 1, 5 (complete section)
⟨declarator⟩, def 2.3
⟨delimiter⟩, def 2.3 synt 2
⟨designational expression⟩, def 3.5.1 synt 3, 4.3.1, 5.3.1 text 3.5.3
⟨digit⟩, def 2.2.1 synt 2, 2.4.1, 2.5.1
dimension, text 5.2.3.2
divide / ÷, synt 2.3, 3.3.1 text 3.3.4.2
do, synt 2.3, 4.6.1
⟨dummy statement⟩, def 4.4.1 synt 4.1.1 text 4.4.3
else, synt 2.3, 3.3.1, 3.4.1, 3.5.1, 4.5.1, text 4.5.3.2
⟨empty⟩, def 1.1 synt 2.6.1, 3.2.1, 4.4.1, 4.7.1, 5.4.1
end, synt 2.3, 4.1.1
entier, text 3.2.5
exponentiation ↑, synt 2.3, 3.3.1 text 3.3.4.3
⟨exponent part⟩, def 2.5.1 text 2.5.3
⟨expression⟩, def 3 synt 3.2.1, 4.7.1 text 3 (complete section)
⟨factor⟩, def 3.3.1
false, synt 2.2.2
for, synt 2.3, 4.6.1
⟨for clause⟩, def 4.6.1 text 4.6.3
⟨for list⟩, def 4.6.1 text 4.6.4
⟨for list element⟩, def 4.6.1 text 4.6.4.1, 4.6.4.2, 4.6.4.3
⟨formal parameter⟩, def 5.4.1 text 5.4.3
⟨formal parameter list⟩, def 5.4.1
⟨formal parameter part⟩, def 5.4.1
⟨for statement⟩, def 4.6.1 synt 4.1.1, 4.5.1 text 4.6 (complete section)
⟨function designator⟩, def 3.2.1 synt 3.3.1, 3.4.1 text 3.2.3, 5.4.4
go to, synt 2.3, 4.3.1
⟨go to statement⟩, def 4.3.1 synt 4.1.1 text 4.3.3
⟨identifier⟩, def 2.4.1 synt 3.1.1, 3.2.1, 3.5.1, 5.4.1 text 2.4.3
⟨identifier list⟩, def 5.4.1
if, synt 2.3, 3.3.1, 4.5.1
⟨if clause⟩, def 3.3.1, 4.5.1 synt 3.4.1, 3.5.1 text 3.3.3, 4.5.3.2
⟨if statement⟩, def 4.5.1 text 4.5.3.1
⟨implication⟩, def 3.4.1
integer, synt 2.3, 5.1.1 text 5.1.3
⟨integer⟩, def 2.5.1 text 2.5.4
label, synt 2.3, 5.4.1

⟨label⟩, def 3.5.1 synt 4.1.1, 4.5.1, 4.6.1 text 1, 4.1.3
⟨left part⟩, def 4.2.1
⟨left part list⟩, def 4.2.1
⟨letter⟩, def 2.1 synt 2, 2.4.1, 3.2.1, 4.7.1
⟨letter string⟩, def 3.2.1, 4.7.1
local, text 4.1.3
⟨local or own type⟩, def 5.1.1 synt 5.2.1
⟨logical operator⟩, def 2.3 synt 3.4.1 text 3.4.5
⟨logical value⟩, def 2.2.2 synt 2, 3.4.1
⟨lower bound⟩, def 5.2.1 text 5.2.4
non-local, text 4.1.3
minus −, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
multiply ×, synt 2.3, 3.3.1 text 3.3.4.1
⟨multiplying operator⟩, def 3.3.1
⟨number⟩, def 2.5.1 text 2.5.3, 2.5.4
⟨open string⟩, def 2.6.1
⟨operator⟩, def 2.3
own, synt 2.3, 5.1.1 text 5, 5.2.5
⟨parameter delimiter⟩, def 3.2.1, 4.7.1 synt 5.4.1 text 4.7.7
parentheses ( ), synt 2.3, 3.2.1, 3.3.1, 3.4.1, 3.5.1, 4.7.1, 5.4.1 text 3.3.5.2
plus +, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
⟨primary⟩, def 3.3.1
procedure, synt 2.3, 5.4.1
⟨procedure body⟩, def 5.4.1
⟨procedure declaration⟩, def 5.4.1 synt 5 text 5.4.3
⟨procedure heading⟩, def 5.4.1 text 5.4.3
⟨procedure identifier⟩, def 3.2.1 synt 3.2.1, 4.7.1, 5.4.1 text 4.7.5.4
⟨procedure statement⟩, def 4.7.1 synt 4.1.1 text 4.7.3
⟨program⟩, def 4.1.1 text 1
⟨proper string⟩, def 2.6.1
quantity, text 2.7
real, synt 2.3, 5.1.1 text 5.1.3
⟨relation⟩, def 3.4.1 text 3.4.5
⟨relational operator⟩, def 2.3, 3.4.1
scope, text 2.7
semicolon ;, synt 2.3, 4.1.1, 5.4.1
⟨separator⟩, def 2.3
⟨sequential operator⟩, def 2.3
⟨simple arithmetic expression⟩, def 3.3.1 text 3.3.3
⟨simple Boolean⟩, def 3.4.1
⟨simple designational expression⟩, def 3.5.1
⟨simple variable⟩, def 3.1.1 synt 5.1.1 text 2.4.3
space ␣, synt 2.3 text 2.3, 2.6.3
⟨specification part⟩, def 5.4.1 text 5.4.5
⟨specificator⟩, def 2.3
⟨specifier⟩, def 5.4.1
standard function, text 3.2.4, 3.2.5
⟨statement⟩, def 4.1.1, synt 4.5.1, 4.6.1, 5.4.1 text 4 (complete section)
statement bracket, see: **begin end**
step, synt 2.3, 4.6.1 text 4.6.4.2
string, synt 2.3, 5.4.1
⟨string⟩, def 2.6.1 synt 3.2.1, 4.7.1 text 2.6.3
string quotes ⌜ ⌝, synt 2.3, 2.6.1 text 2.6.3
subscript, text 3.1.4.1
subscript bound, text 5.2.3.1
subscript brackets [ ], synt 2.3, 3.1.1, 3.5.1, 5.2.1
⟨subscript expression⟩, def 3.1.1 synt 3.5.1
⟨subscript list⟩, def 3.1.1
⟨subscripted variable⟩, def 3.1.1 text 3.1.4.1
successor, text 4
switch, synt 2.3, 5.3.1, 5.4.1
⟨switch declaration⟩, def 5.3.1 synt 5 text 5.3.3
⟨switch designator⟩, def 3.5.1 text 3.5.3
⟨switch identifier⟩, def 3.5.1 synt 3.2.1, 4.7.1, 5.3.1
⟨switch list⟩, def 5.3.1
⟨term⟩, def 3.3.1
ten $_{10}$, synt 2.3, 2.5.1
then, synt 2.3, 3.3.1, 4.5.1
transfer function, text 3.2.5
true, synt 2.2.2
⟨type⟩, def 5.1.1 synt 5.4.1 text 2.8
⟨type declaration⟩, def 5.1.1 synt 5 text 5.1.3
⟨type list⟩, def 5.1.1
⟨unconditional statement⟩, def 4.1.1, 4.5.1
⟨unlabelled basic statement⟩, def. 4.1.1
⟨unlabelled block⟩, def 4.1.1
⟨unlabelled compound⟩, def 4.1.1
⟨unsigned integer⟩, def 2.5.1, 3.5.1
⟨unsigned number⟩, def 2.5.1 synt 3.3.1
until, synt 2.3, 4.6.1 text 4.6.4.2
⟨upper bound⟩, def 5.2.1 text 5.2.4
value, synt 2.3, 5.4.1
value, text 2.8, 3.3.3
⟨value part⟩, def 5.4.1 text 4.7.3.1
⟨variable⟩, def 3.1.1 synt 3.3.1, 3.4.1, 4.2.1, 4.6.1 text 3.1.3
⟨variable identifier⟩, def 3.1.1
while, synt 2.3, 4.6.1 text 4.6.4.3

This revised report is reprinted by permission of the International Federation for Information Processing, who ask us to state that reproduction of the whole text only is permitted without formality.