3201
7-27-61

ORNL-3148
UC-32 – Mathematics and Computers

STRUCTURE AND USE OF ALGOL 60

Hermann Bottenbruch

OAK RIDGE NATIONAL LABORATORY
operated by
UNION CARBIDE CORPORATION
for the
U.S. ATOMIC ENERGY COMMISSION

ORNL-3148

Contract No. W-7405-eng-26

MATHEMATICS PANEL

STRUCTURE AND USE OF ALGOL 60

Hermann Bottenbruch

DATE ISSUED

JUL 26 1961

OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee
operated by
UNION CARBIDE CORPORATION
for the
U. S. ATOMIC ENERGY COMMISSION

THIS PAGE

WAS INTENTIONALLY

LEFT BLANK

CONTENTS

ABSTRACT

ALGOL 60 is a universal, algebraic, machine-independent programming language. It was designed by a group representing computer societies from many different countries. Its primary aims are:

1. Simplification of program preparation.
2. Simplification of program exchange.
3. Incorporation of the important programming techniques presently known.

The ALGOL 60 language is defined in *Communications of the ACM* **3**, 299–314 (1960). The present report is an elaboration of the concepts of ALGOL 60, mostly with the help of illustrative examples. It is intended for people who are familiar with the general ideas of programming and mathematical notation.

STRUCTURE AND USE OF ALGOL 60

Hermann Bottenbruch[1]

INTRODUCTION

ALGOL was designed as a language to be used for the description of computing processes in a machine-independent way, and in this respect it resembles other languages such as FORTRAN, MATHE-MATIC, and IT. ALGOL is an international effort, and we hope that it will become the main vehicle for communication of algorithms in publications and reports. The same aims which governed the design of the aforementioned systems governed the design of ALGOL. The following is cited from the ALGOL 58 report (*1*):[2]

1) The language should be as close as possible to standard mathematical notation and be readable with little further explanation.
2) It should be possible to use it for the description of computing processes in publications.
3) The new language should be mechanically translatable into machine programs.

However, in ALGOL, ease of expressing a computational process and consistency with existing mathematical notation were stressed more than in other systems, and so ALGOL is somewhat further away from "thinking in terms of a computing machine" and closer to "thinking in terms familiar to human beings" than other systems. This difference shows up more in the elaborate parts of the language than in the simple parts.

The ALGOL 60 language[3] is defined in a very precise and concise way in "Report on the Algorithmic Language ALGOL 60" by Peter Naur (editor), *et al.* See (*2*). The present paper is an elaboration of the concepts defined in this report. It is not a definition of the language, because it does not give all the composition rules. It explains, rather, the meaning and proper use of the devices provided by ALGOL 60. The main vehicle in accomplishing this task is the discussion of illustrative examples. Some of the concepts, especially "block," "procedures," "local or own quantities," are useful only in programs

[1]On leave of absence from Institute für Praktische Mathematik, Technische Hochschule, Darmstadt.

[2]Numbers in parentheses refer to the corresponding items in the bibliography.

[3]ALGOL 58 (*1*) was a preliminary language. ALGOL 60 is the ALGOL language, which was based on the ALGOL 58 report and discussions of it in the *Communications of the ACM* and in the *ALGOL Bulletin.*

which are so large that the explanation of the logic of the program would overshadow the explanation of the language. Therefore the examples are sometimes oversimplified so that they are not practically useful except for the purpose of explanation, and some of the examples are only skeletons, in that they leave out certain parts of the program, showing only those which are pertinent to the explanation.

Although it is the main aim of this paper to explain ALGOL as far as writing correct programs is concerned, some effort is made to show how efficient programs may be written. The remarks concerning storage allocation in blocks (see p 25) fall into this category.

A SAMPLE PROGRAM IN ALGOL 60

The following program computes the expression

$$E = \sum_{s=0}^{10} \frac{1}{s!\,(n+s)!} \cdot \left(\frac{x}{2}\right)^{2\cdot s+n}$$

This expression, by the way, represents the first 11 terms of the power series expansion for the modified Bessel function of integer order ≥ 0. This fact is not important in order to understand the example. The expression cannot be evaluated for $x = n = 0$, because for $s = 0$ we get the undefined expression 0^0. In the case $x = n = 0$ the program will give the result 1.

The program given below will serve to give a rough impression of the features of the language, and by comparing the program with the formula, some of the notational peculiarities become clear. The program is not complete, insofar as it does not contain "declarations" (see pp 20 ff.).

The program given is in no way an efficient formulation for the computation of this expression, and many refinements are possible by eliminating unnecessary computational steps. Such refinements, however, would not serve our present purpose of showing some of the features of the language.

The program reads in four values from an input medium and assigns these values to the variables named *x*0, *deltax*, *xmux*, *n*. Then it computes E for x = *x*0, *x*0 + *deltax*, *x*0 + 2 × *deltax*, . . . , *xmax*, and punches x, n, and E on an output medium:

```
                                                              Line
M  :  read x0, deltax, xmax, n ;                                1
      x := x0 ;                                                 2
      nfac := 1 ;                                               3
      for i := 2 step 1 until n do nfac := nfac × i ;           4
      if x = 0 ∧ n = 0 then begin sum := 1 ; go to P end ;      5

L  :  denom := nfac ;                                           6
      sum := 0 ;                                                7
      for s := 0 step 1 until 10 do                             8
             begin sum := sum + (x/2) ↑ (n + 2 × s)/denom ;     9
                   denom := denom × (s + 1) × (s + 1 + n)      10
             end ;

P  :  punch x, n, sum ;                                        11
      x := x + deltax ;                                        12
      if x ≦ xmax then go to L ;                               13
      go to M ;                                                14
```

Notes

1. The program is a succession of *statements*, which are normally executed in the order in which they are written down. Some of the statements have *labels* in front of them (line 1, 6, 11) which serve as marks and which are used in *go to statements* (line 5, 13). Go to statements interrupt the normal sequence of execution of the statements and indicate the next statement to be executed.

2. The symbols M, L, P are used as names of statements. The symbols *x0*, *deltax*, *xmax*, n, x, *nfac*, *denom*, *sum*, s, i are used as names of *variables*. These names are chosen arbitrarily, and replacing a name consistently throughout the program by a name different from all the other names used does not change the meaning of the program. The names are, however, somehow suggestive of the quantities which they denote. In contrast to these *free names* there are certain other words used in the program like **read**, **step**, **until**, **go to**, **begin**, **end**, **punch**. These are fixed constituents of the language. These fixed words are always printed in bold face and in the discussions below are treated as a single symbol, such as +, −, etc.

3. The meaning of the statement in line 1 is obvious from the description preceding the program.

4. The statements in lines 2, 3, 6, 7, 9, 10, 12 are *assignment statements*. The value of the *expression* on the right side of the symbol := is computed and assigned to the variable at the left side of that symbol. Assignment statements are also part of the *conditional statement* in line 5 and of the *for statement* in line 4.

5. The statement in line 4 is a *for statement*, which has as one of its constituents another statement ($nfac := nfac \times i$), which in this particular case is an assignment statement. This constituent statement is executed for certain values of the *loop variable* i which are obvious from the *for list element* 2 **step** 1 **until** n. If the upper limit n of this for list element is less than 2, the loop will not be executed at all. The net result of statements 3 and 4 together will be that the variable $nfac$ has the value $n!$. The phrase "**for** i := 2 **step** 1 **until** n **do**" is called a *for clause*.

6. The statement in line 5 is a *conditional statement*. It contains as one of its constituents another statement, which in this case is the *compound statement*: **begin** sum := 1; **go to** P **end**. Execution of a compound statement means execution of all the statements constituting the compound statement. The statement is, however, only executed if the condition following the **if** of the conditional statement is true. Otherwise the complete conditional statement is void. The condition itself is compounded of the two conditions $x = 0$ and $n = 0$ by means of the *Boolean operation* $\wedge$ (logical and), which means that the compounded condition is true if both the constituent conditions are true. In the sequel, we will use the word *Boolean expression* instead of *condition*.

7. The three lines 8, 9, and 10 together form a single statement (a for statement, see note 5), where the statement governed by the for clause is again a compound statement, since in this case the two statements in lines 9 and 10 are to be executed for the designated values of s.

8. (For purists and puristic critics).[4] ALGOL 60 does not contain any statements for *input and output*. Such statements, of course, have to be added to the language if one wants to write any useful programs.

9. The concepts of notes 1 to 8 which are printed in italics are, in addition to others, more fully explained in the following sections.

[4]A puristic critic is somebody who criticizes ALGOL on the fact that a translator is useless which fulfills ALGOL to the last atom of printer's ink.

DETAILED DISCUSSION OF SOME SIMPLE ALGOL CONCEPTS

The definition of an ALGOL concept consists of the syntactic part and the semantic part. In the syntactic part we describe which sequences of ALGOL symbols may represent the concept. In the semantic part we describe what the meaning of this concept is.

The ALGOL 60 report gives the syntactic part in a very rigorous form. We do not aim at such rigor. One way of "defining" a concept which is used in the present exposition is simply to give examples of strings which do represent a concept and of those which do not. Although this sometimes leaves some doubt, it is hoped that examples and counterexamples make the general idea sufficiently clear. Clarifications are added in common language wherever this is desirable. Even so, some exceptional cases may not be covered by the description given here, and for completeness the reader is referred to the ALGOL 60 report. For the "normal" user of ALGOL, however, it should not be necessary to go through the painstaking process of disentangling the definitions in the report after reading this introduction, which stresses the more frequent and useful concepts while treating the less important features with less care.

As far as the "semantic part" of the definition is concerned, we have omitted it if the meaning is obvious from common usage.

Arithmetic Expressions and Their Constituents: Numbers, Variables, Operations, Special Functions

Arithmetic expressions are formed in almost the usual way, with a few notational peculiarities. The quantities involved in forming an expression are *numbers*, *simple* or *subscripted variables*, and *functions* (see Table 1).

arithmetic expressions

numbers

simple variables

Simple variables are denoted by *identifiers*.

identifier

An identifier is any string of letters and digits of any length beginning with a letter. Capital and lower-case letters may be used.

Subscripted variables are denoted by an identifier followed by a list of expressions enclosed in brackets.

subscripted variables

There are no restrictions on the expressions used in subscript positions. Subscripted variables are defined only for integer subscripts. However, even the evaluation of a simple expression like $(n)\times(n-1)/2$ may yield a noninteger number due to round-off. Therefore the following convention was adopted:

If the value of an expression used in a subscript position is not an integer, then this value is changed to the nearest integer in the sense of proper round-off. Thus: $a[0.6] = a[1.4] = a[1.479] = a[0.5] = a[1]$, $a[1.5] = a[2]$, $a[-1.5] = a[-1]$.

noninteger subscript expressions

Table 1. Examples and Counterexamples of Arithmetic Expressions and Their Constituents

Examples	Counterexamples	Structure
1 2 −4.8 −0.38 0.38 $-{}_{10}-4$[a] $-{}_{10}4$ $_{10}4$ $_{10}+5$ $+{}_{10}5$	3×4 π a 127. $\sqrt{6}$ $127.{}_{10}4$ $_{10}0.5$ $_{10}2.8$	Numbers
a *Soup* *a1* *a2* *A1* *A2* *alpha* *alpha1* *astar* *pi*	π $a[1]$ a_1 2 $2a$ α a^{-1} $a\times$	Identifiers
	See identifiers	Simple variables
a $a[1]$ $a[i[1]]$ $a[i[1] + 3 \times n]$ $b[1, 2]$ $b[n \times j[i] + a, r + 2 \times pi]$ $a[\sin(2 \times pi \times j/3)]$	a_1 $a_{2,3}$ π $-\alpha$	Variables, simple or subscripted
3×4 $3.1 \times 4 \times sin(3 \times pi/5)$ $a \uparrow (-2)$ $a \uparrow 2$ $a \uparrow 2 + b \uparrow 2$ $a \uparrow (2 + b \uparrow 2)$ $sqrt(6)$ $a \uparrow 2 \uparrow 3$ meaning $(a^2)^3$ $a \uparrow (2 \uparrow 3)$ meaning $a^{(2^3)}$	$a \uparrow -2$ $\sqrt{6}$ 2π $2\,pi$	Arithmetic expression[b]

[a]The symbol $_{10}$ has the meaning $\times\ 10 \uparrow$, for example, $_{10}4 = 10^4$; $0.41{}_{10}-2 = 0.41 \times 10^{-2} = 0.0041$; it is used in almost the same way that the letter *E* is used in FORTRAN numbers.

[b]For an extension of the notion of expression, see "Functions," p 37. For conditional expressions, see "Peculiarities," p 22.

The totality of subscripted variables denoted by the same identifier is called an *array*. See *array declarations*, p 21. Permissible operations in forming arithmetic expressions are: the binary operations addition, subtraction, multiplication, division, exponentiation, and integer division, denoted by +, −, ×, /, ↑, and ÷; the unary operations + and −; and the special functions *abs, sign, sqrt, sin, cos, arctan, ln, exp, entier*. The definitions of *sign* and *entier* are given below.

arrays operations

special functions

The exponent part of the operation ↑ stands on the same line as the base number and must therefore be enclosed in parentheses if it is an expression containing an operation. See p 21 for integer division.

Precedence of operations is understood in the usual way; that is, ×, ÷, and / are "stronger" than binary plus and minus; ↑ is stronger than all other binary operations; unary operations are stronger than binary operations. If precedence is not specified by these rules, the operation farther to the left is stronger. Examples are:

precedence of operations

$a + b \times c$ means $a + (b \times c)$

$a + b - c$ means $(a + b) - c$

$a - b + c$ means $(a - b) + c$ and not $a - (b + c)$

$a + b \uparrow c + d$ means $a + (b \uparrow c) + d$

$b \uparrow -c$ is not permissible

$a / b \times c$ means $(a / b) \times c$ and not $a/(b \times c)$

$-a \uparrow b$ means $(-a) \uparrow b$ and not $-(a \uparrow b)$

Expressions which are ambiguous in usual notation, such as the parentheses-free form of the last two examples, should be avoided by proper placement of parentheses.

It should not be overlooked that the associative law does not hold in rounded-off computations. In some cases it may be important whether the expression $a - b + c$ is executed as $(a - b) + c$ or as $a - (b - c)$, for example, if c is small compared with a and b, and a and b are almost equal and contain almost equal errors. If it is important that the expression be evaluated as $(a - b) + c$, this should be indicated by parentheses despite the fact that ALGOL implies this interpretation.

The meaning of the special functions apart from *sign* and *entier* is obvious. The function *sign* is defined by

sign and entier

$$sign\ (E) = \begin{matrix} +1 & \text{if} & E > 0 \\ 0 & \text{if} & E = 0 \\ -1 & \text{if} & E < 0 \end{matrix}$$

The function *entier* (E) is defined to give the largest integer not greater than E. Thus *entier* (1) = 1, *entier* (−1) = −1, *entier* (−1.1) = −2, *entier* (1.1) = 1.

Arithmetic Assignment Statements

An *arithmetic assignment statement* is of the form

arithmetic assignment statement

$$\mathbf{V} := \mathbf{E}\ ,$$

where **V** stands for a variable and **E** stands for an arithmetic expression.

Examples are

$$a := 1$$
$$i := i + 1$$
$$a := b$$
$$q[j] := r[s + t/2]$$

A *statement* in general is a rule to perform some action. The action denoted by an arithmetic assignment statement is (1) the computation of the value of an expression and (2) the assignment of that value to the variable. The "value associated with a variable" is the last value assigned to that variable.

The formula "**V** := **E**" accompanied by the qualifying remarks "where **V** is a variable and **E** is an arithmetic expression" is an example of what will be called a *syntactic skeleton.* In particular, **V** := **E** is the syntactic skeleton for an arithmetic assignment statement. If, in this skeleton, **V** is replaced by a variable and **E** by an arithmetic expression, the result is a syntactically correct arithmetic assignment statement.

The bold-face capital letters **V** and **E** are variables[5] which stand for ALGOL *structures.* In general, a syntactic skeleton is a formula which is made up from ALGOL symbols [except digits and (italic) letters] and bold-face letters. In most cases such a formula is followed by remarks that identify the structures for which the bold-face letters stand. If one replaces each bold-face letter by any ALGOL structure for which it stands, the result will be an ALGOL structure of the type designated by the skeleton. Since the structures variable, arithmetic expression, and statement occur so frequently in syntactic skeletons, we introduce the following convention:

> Whenever the bold-face letters **V**, **E**, or **S**, with or without subscripts, appear in a syntactic skeleton, they stand for the structures variable, arithmetic expression, or statement respectively.

Boolean Expressions

Boolean expression

Boolean expressions are used to form a truth value, that is, an entity which is either true or false. A Boolean expression is formed

[5]In order to distinguish these variables from ordinary ALGOL variables, they are sometimes called "meta-variables." They never appear in ALGOL programs; they are only used when talking *about* ALGOL programs. Sometimes they are called "syntactic variables."

with the following quantities: (*a*) the truth values **true** and **false**, (*b*) Boolean simple or subscripted variables (see "Declarations and Blocks," p 20, and (*c*) arithmetic comparisons. Each of these quantities designates a truth value, which is obvious in case (*a*). In case (*b*) it is the last truth value assigned to the variable (see Boolean assignment statement). In case (*c*) it is determined by the result of the comparison.

arithmetic comparison

An *arithmetic comparison* is of the form

E R E,

where **R** stands for any of the six relations $=, \neq, >, \geqq, <, \leqq$. Some examples are:

$$1 \leqq 2 , \qquad a + b \leqq c \times (d + e - f \times g) ;$$

$$1.34 \leqq 2 \times \mathrm{sqrt}\ (3.12) .$$

The truth value of comparisons 1 and 3 is always true, whereas the truth value of comparison 2 depends on the values of *a*, *b*, . . . , *g*.

precedence of operations in Boolean expressions

Out of these constituents, Boolean expressions can be formed according to the rules of Boolean algebra. The permissible Boolean operations are "not," "and," "inclusive or," "implies," "equivalent," denoted by $\neg$, $\wedge$, $\vee$, $\supset$, $\equiv$.[6] The rules of precedence are given by the order in which the operations are listed. Thus $p \vee q \wedge r$ means $p \vee (q \wedge r)$, and $p \wedge q \vee r$ means $(p \wedge q) \vee r$. If precedence is not determined by parentheses and the above mentioned rules, operations are carried out from left to right. Thus $a \equiv b \equiv c$ means $(a \equiv b) \equiv c$, which sometimes has a truth value different from $a \equiv (b \equiv c)$. A few examples for Boolean expressions are:

$$x \neq -2 \vee x = -2, \textbf{true}, \neg\ \textbf{true} \equiv \textbf{false}, a \leqq b \wedge c \leqq d \wedge e ,$$

$$\neg (a \leqq b \equiv -a \leqq -b) , \qquad A[i + 3] \leqq B[i, 2] ,$$

$$a \leqq b \wedge b \leqq c \equiv a \leqq c .$$

The last expression is equal to

$$[(a \leqq b) \wedge (b \leqq c)] \equiv (a \leqq c) .$$

Some of the Boolean expressions given in the example are true regardless of the values of the variables appearing in them. The identifiers

[6]The result of the Boolean operation "and" is true if both operands are true. The result of the Boolean operation "or" is true if one of the operands is true. The result of the Boolean operation "implies" is true either if the second operand is true or if the first operand is false. The result of the Boolean operation "equivalent" is true either if both operands are true or if both operands are false. In the cases not mentioned the result is false.

a, b, c, d, i, x, denote real or integer variables (see "Declarations and Blocks," p 20). The identifier *e* denotes a Boolean variable.

The most common form of Boolean expressions are arithmetic comparisons. The program on p 3 contains in line 5 an expression composed of the two comparisons $x = 0$ and $n = 0$. The program on p 29 contains in line 5 a Boolean expression composed of three comparisons. Another example is in the merge procedure on p 45 in the line labeled *Q*. More complicated Boolean expressions occur in programs with a complicated structure, for example, compilers.

Boolean variables are useful for "storing" the results of arithmetic comparisons in a readily accessible way.

Boolean Assignment Statements

Boolean assignment statement

Boolean assignment statements are denoted by

$$\mathbf{V} := \mathbf{B} \; ,$$

where **V** is a Boolean variable and **B** is a Boolean expression. This statement assigns the truth value of **B** to the variable **V**.

Conditional Statements

conditional statement

A *conditional statement* is denoted by

$$\mathbf{if\ B\ then\ S} \; ,$$

or alternatively by

$$\mathbf{if\ B\ then\ S_1\ else\ S_2} \; ,$$

where **B** is a Boolean expression and **S**, $\mathbf{S_1}$, $\mathbf{S_2}$ are statements. Execution of a conditional statement of the first form means execution of statement **S** if the truth value of **B** is true. Otherwise the statement is void. Execution of a conditional statement of the second form means execution of $\mathbf{S_1}$ if **B** is true and execution of $\mathbf{S_2}$ if **B** is false. The statement following the **then** must not be a conditional statement. The statement following the **else** may be a conditional statement, thus allowing constructions of the kind:

$$\mathbf{if\ B_1\ then\ S_1\ else\ if\ B_2\ then\ S_2\ else\ if\ B_3\ then\ S_3} \; ,$$

where $\mathbf{S_3}$ may again be a conditional statement.

Compound Statements

compound statement, begin end

A sequence of one or more statements, separated by semicolons, and enclosed in the so-called *statement parentheses* **begin** and **end** is called a *compound statement*. A compound statement is also a statement, and may therefore be a constituent of another compound statement. An example of a compound statement is:

begin $a := b$; $x := y$; $n := v$ **end**

Examples of conditional statements, each of which has a compound statement as one of its constituents are:

if $a \leqq 2$ **then begin** $a := b$; $x := y$ **end else** $u := 0$

if $a \leqq 2$ **then** $a := b$ **else begin** $x := y$; $u := 0$ **end**

In the first conditional statement the statements $a := b$ and $x := y$ are executed if $a \leqq 2$. Otherwise the statement $u := 0$ is executed. In the second conditional statement the statement $a := b$ is executed if $a \leqq 2$. Otherwise the statements $x := y$ and $u := 0$ are executed.

Use of compound statements in conditional statements can always be avoided by means of go to statements and labels. For example, the first conditional statement above is equivalent to

if $a \leqq 2$ **then go to** L ;

$u := 0$; **go to** M ;

L : $a := b$; $x := y$;

M : (next statement) .

This explicit form is bad ALGOL style.

Loops

A loop is a device which facilitates the repeated execution of a statement for different values of a distinguished variable, the so-called loop variable. A simple example of a loop is: loop

for $i := 1$ **step** 1 **until** 10 **do** $a[i] := i \uparrow 2$.

This loop assigns the value 1 to the variable $a[1]$, 4 to the variable $a[2]$, . . . , 100 to the variable $a[10]$. The values which are assigned to the loop variable are in general determined by so-called "for list elements" (see below). A loop is a statement, and the ALGOL term for this is *for statement*. A for statement has one of the two forms: for statement

for V := FL do S

for V := $\mathbf{FL}_1$, $\mathbf{FL}_2$, . . . , $\mathbf{FL}_n$ do S

where **FL**, $\mathbf{FL}_1$, ..., $\mathbf{FL}_n$ designate *for list elements.* A for list element has one of the three forms:

for list elements

$\mathbf{E}_1$ **step** $\mathbf{E}_2$ **until** $\mathbf{E}_3$

E while B

E

Each for list element designates a sequence of values to be assigned to the loop variable **V**. The first element, for example, designates those elements of the sequence $\mathbf{E}_1$, $\mathbf{E}_1 + \mathbf{E}_2$, $\mathbf{E}_1 + \mathbf{E}_2 + \mathbf{E}_2$, ... which lie between $\mathbf{E}_1$ and $\mathbf{E}_3$, inclusive. A loop may be expressed equivalently by a sequence of ALGOL statements not containing loops. The following examples make this sufficiently clear:

1. **for** $i := a$ **step** 1 **until** b **do** $c[i] := d[i]$ is equivalent to

```
    i :=a ;
L : if i ≤ b then begin c[i] := d[i] ; i := i + 1; go to L end
```

Note that the loop is void in case $b < a$. In for list elements with negative step-expression the comparison must use $\geq$ instead of $\leq$.

2. **for** $i := i + 1$ **while** $a[i - 1] > d$ **do** $a[i] := a[i - 1]/i$ is equivalent to

```
L : i := i + 1 ;
    if a[i - 1] > d then begin a[i] := a[i - 1]/i ; go to L end
```

3. **for** $i := a + b$ **do** $x[i] := y[i]$ is equivalent to

```
    i := a + b ;
    x[i] := y[i]
```

Loops with more than one for list element are equivalent to a sequence of loops with only one for list element. Thus:

for statements with several for list elements

for V := $\mathbf{FL}_1$, $\mathbf{FL}_2$, $\mathbf{FL}_3$ **do S** is equivalent to

for V := $\mathbf{FL}_1$ **do S** ;

for V := $\mathbf{FL}_2$ **do S** ;

for V := $\mathbf{FL}_3$ **do S**

Conditional statements and for statements are both structures which may be parts of other statements and which have statements as their constituents. In order to indicate the logical dependence of these statements, one should use statement parentheses even where they are not indispensable. In the example on p 43, one would not need the begin and end because only one statement is governed by the for clause. If this statement were part of another conditional statement the begin and end are necessary in order to show the if clause to which the else belongs. Example:

```
p := 1 ;
if e > 2 then
    for p1 := (e + p) ÷ 2 while e − p ≠ 1 do
        begin if a[p1] < b then e := p1 else p := p1 end
```

Without the statement parentheses this program could be interpreted in the following way:

```
p := 1
if e > 2 then
    for p1 := (e + p) ÷ 2 while e − p ≠ 1 do
        begin if a[p1] < b then e := p1 end ;
else p := p1
```

Labels, go to Statements

labels, go to statements

Any identifier and any unsigned integer may serve as a *label*. Labels and *go to statements* are described in the example, p 3, note 1. See also "Designational Expression," p 27.

Dummy Statement

dummy statement

A *dummy statement* is represented by "no symbol." A dummy statement executes no operation. It may serve to place a label. An example is the statement with label L in the following program to find the maximum of $a[1]$ to $a[n]$:

```
imax := 1 ; M := a[1] ;
for i := 1 step 1 until n do
    begin if a[i] < M then go to L
                    else begin imax := i ;
                               M := a[i]
                         end ;
L : end
```

Input and Output of Information

There are no statements for input and output of information in the ALGOL language. The statements described below are used to supplement the language in this respect. They are, along with others, used for input and output of information in the Oracle ALGOL Translator (*3*). The statements given below constitute a bare minimum of input-output.

commands and thus are not intended as a proposal for general acceptance. Such a proposal must contain some means for format designation and input and output of alphameric information. Some practically useful minimal set of statements would be desirable as a general standard.

read

The statement **read V** means: Read the next number from input tape, convert it to internal representation, and assign it to the variable **V**. The number read in is supposed to be punched in ALGOL form on an input medium. Any ALGOL symbol not compatible with the structure of an ALGOL number terminates the number.

The statement **read** $V_1, V_2, \ldots, V_n$ is equivalent to the sequence of statements:

read V_1 ; **read** V_2 ; . . . ; **read** V_n

punch

The statement **punch E** has the meaning: Compute the value of the expression **E** and punch it on the output medium.

The statement

punch $E_1, E_2, \ldots, E_n$; is equivalent to

punch E_1 ; **punch** E_2 ; . . . ; **punch** E_n

carriage return

The statement **carriage return** activates the punching of a carriage return and line feed symbol (or equivalent) on the output medium.

For further remarks on input-output see strings, p 15, and machine code procedures, p 39.

Comment

comment

There is the possibility of inserting into a program *comment*, or *text*, which does not affect the meaning of the program, but which helps to make the meaning of a program clear to the (human) reader. The comment rules are:

1. The symbol **comment** and any character between this symbol and the first semicolon after **comment** are text.
2. Any character between the symbol **end** and the first semicolon or **end** or **else** following this **end** is text. Some examples are:

comment $a + b - c$ is positive at this point ;

comment if the program ever gets to this point there is a mistake in the input data. Do not worry ;

if $a = b$ **then begin** $x := g$; $y := z$ **end**. This is a very exceptional case **else** $x := g + 1$;

In the last example the symbols beginning with the period after the **end** and ending with the word "case" are text.

The second comment convention is often used to "mark" an end in a way which facilitates finding the matching begin. In case an end terminates a loop, one can repeat the loop variable after the end. If the compound statement has a label, one can repeat the label of this statement after the end. It has been the experience of this writer that the following way of writing compound statements displays the structure of long compound statements more effectively: Write matching begins and ends either in the same row or in the same column of the program. Subordinate begins and ends are indented farther to the right.

ways of indicating matching begins and ends

Strings

An ***ALGOL*** *string* is either a sequence of ALGOL symbols enclosed in the opening and closing *string quotes* ' '; or it is a sequence of ALGOL symbols and ALGOL strings enclosed in these string quotes. Examples are:

string quotes

'a b c d'

'ab 'cde' fg'

Strings cannot be used in ALGOL 60 proper. Strings are useful in input-output operations. With strings in mind it would be useful to interpret output operations in the following way: Whenever a string appears in an output statement at the place of an expression, this means that the elements of the string are put on the output medium. With this general interpretation the statement

punch '*A* =', 1.34, '##*B* =', *B*;

would place the following symbols on the output medium:[7]

A = 1.34 ,   *B* = ... ,

where . . . . stands for the value of the variable *B* at the time of execution of this statement. In order to make strings really useful, one would need string variables, string assignments, and some string operations. String operations can be introduced into ALGOL by means of procedures written in machine code (see p 39).

Punctuation

Proper use of punctuation symbols hardly represents any difficulties, with the possible exception of the semicolon. In using the semicolon one should have the following facts in mind: A semicolon is used only to separate the different statements of a compound statement. It is not part of a conditional statement or of a for statement, except when these have a compound statement as one of their constituents. From this it may be seen that the sequence

semicolon

[7]The # is the ALGOL representation of "blank." Blanks are normally disregarded in ALGOL, but they are meaningful in strings.

"; else" can never occur in an ALGOL program, and that in the sequence "; end" the semicolon is redundant. Example 1:

```
if $a \leq b$ then $x := y$ else $x := z$
```

Insertion of a semicolon after the y would terminate the conditional statement, leaving the else without an if clause. An "intelligent" translation program would be prepared for that mistake which results from the assumption that a statement must be terminated by a semicolon, and it would simply eliminate the semicolon, giving an appropriate error message. Example 2:

```
if $a \leq b$ then begin $x := y$ ; $y := z$ ; end ;
```

The semicolon after the z is redundant.

Statements

Thus far we have discussed the following types of statements:

1. assignment statements (arithmetic and Boolean),
2. go to statements,
3. for statements,
4. conditional statements,
5. read, punch, and carriage return statements,
6. compound statements.

The notion of statement is recursive insofar as some of the constituents of for statements, conditional statements, and compound statements are themselves statements.

labeled statements

Any statement may be preceded by one or several labels, each followed by a colon. A statement together with its labels is again a statement of the same type. For example, a conditional statement preceded by a label is still a conditional statement.

conditional statement enclosed in begin end

A conditional statement enclosed in the statement parentheses **begin** and **end** is no longer a conditional statement. The first symbol of an unlabeled conditional statement is always **if**.

blocks and procedure statements; see p 20 and 28

There are only two more types of statements in addition to those discussed so far, namely, *procedure statements* and *blocks*. They are discussed in subsequent sections.

Examples of ALGOL Programs

1. The reader is referred to the example given on p 3.
2. Multiplication of a matrix A by a matrix B to form a matrix C:

```
for $i$ := 1 step 1 until $n$ do
  for $k$ := 1 step 1 until $n$ do
    begin $S$ := 0 ;
```

```
                for $j$ := 1 step 1 until $n$ do
                    $S := S + A[i, j] \times B[j, k]$ ;
                $C[i, k] := S$
            end ;
```

3. Sorting a one-dimensional array of numbers $N[1]$, $N[2]$, ... , $N[k]$ according to size by successive interchanges.

Note: This method is slow if k is large. Sorting methods for large arrays are considered in later parts of this report, after the discussion of procedures:

```
for $i$ := 1 step 1 until $k - 1$ until $k - 1$ do
   if $N[i + 1] < N[i]$ then
      for $j := i + 1$, $j - 1$ while $N[j - 1] > N[j] \wedge j \neq 1$ do
         begin $b := n[j - 1]$ ;
               $N[j - 1] := N[j]$ ;
               $N[j] := b$
         end ;
```

It may be of interest to write the last loop in this program in a form not containing for statements:

```
      $j := i + 1$ ;
      $b := N[j - 1]$; $N[j - 1] := N[j]$; $N[j] := b$
  $L$ : $j := j - 1$ ;
      if $N[j - 1] > N[j] \wedge j \neq 1$ then
         begin $b := N[j - 1]$; $N[j - 1] := N[j]$; $N[j] := b$ ;
               go to $L$
         end ;
```

Recursive Definition of ALGOL Concepts

The concepts of ALGOL 60 are defined recursively: A concept C_1 which is used as a constituent in defining C_2 may itself require C_2 as one of its defining constituents. Examples of this are the two concepts subscripted variable and expression. An expression is formed according to the usual rules of arithmetic from numbers and simple and subscripted variables. A subscripted variable, on the other hand, may be formed by means of arithmetic expressions in the subscript positions; see the examples on p 6. Other examples are the concepts statement and conditional statement: To form a conditional statement requires that one or more statements be formed first. On the other hand, since a conditional statement is a statement, these constituent statements may themselves be conditional.

This recursive definition causes some trouble in describing the language and in understanding a description of the language. Thus, in the present description, we introduce the concept of conditional statement on p 10, using the notion of statement only in so far as it has already been explained. Up to this point, the only statements which may be used to construct a conditional statement are arithmetic or Boolean assignment statements and conditional statements. Later we add rules for constructing statements, and without explicitly mentioning it we imply that the expanded notion of statement may be used in all those construction rules which were previously given. In order to get a clear picture of these recursively defined structures, one should read the construction rules several times forward and backward, forming examples of these structures and using these examples in forming other structures according to the construction rules.

This recursive definition of the ALGOL concepts accounts for much of the thought required for building a translator: From a given structure one has to find the rules according to which it is constructed.

In defining the ALGOL language there is only one thing which *must* be clearly defined, and that is the concept of program; that is, which construction rules lead to ALGOL programs, and what is the meaning of the program. All the other concepts, such as expressions, are auxiliary, and the language might well be defined by using some other auxiliary concepts. In this presentation we use many of the auxiliary concepts introduced in the ALGOL 60 report. Some of the concepts which we do not find of sufficient significance (such as *basic statement*, *compound tail*, *Boolean factor*, *simple Boolean*) are not explained or used. Some of the auxiliary concepts are used without explanation, if their meaning is sufficiently clear from ordinary grammar (e.g., *unlabeled statement*).

Peculiarities

1. Multiple Assignment Statement. – This type of statement may be illustrated by examples:

multiple assignment statement

```
a := b := c := d + e means c := d + e ;
                           b := c ;
                           a := b
```

```
a[i] := i := 2 means j := i ;
                     i := 2 ;
                     a[j] := 2
```

The "sneaky" point in this last example lies in the fact that the variable $a[i]$ is determined by the value which i had *before* the execution of the assignment statement [see the ALGOL 60 report (*2*), Sec 4.2].

conditional expression

2. **Conditional Expressions.** – Example:

if $a = b$ **then** 3 **else** 4

The value of this expression is 3 if $a = b$; otherwise the value will be 4. Conditional expressions, if used as operands of other expressions, must be enclosed in parentheses:

a + (**if** $a = b$ **then** 3 **else** 4) + 6

The following construction is possible:

a[**if** $x = y$ **then** 3 **else** 2, **if** $x \leqq y$ **then** 2 **else** 3]

designational expression

3. **Designational Expressions.** – These are treated in connection with switches; see p 26. See also the ALGOL 60 report, Sec 3.5.

expressions in for list elements

4. **Dynamic Interpretation of Expressions in For List Elements.** – The expressions appearing in a for clause may contain the loop variable, or other variables, the value of which is changed in the statement governed by the clause. Whenever the evaluation of such an expression is called for in the execution of the for statement, the present values of these variables will be used for the evaluation. This means that in general these expressions must be evaluated each time the statement governed by a for list element is executed. If the value of these expressions remains constant during execution of the for statement, their values need be computed only once. It is not easy for a translator to determine in which category a for statement belongs. The programmer can "help" the translator in producing an efficient machine program by using single variables or constants in for list elements where this is possible.

Example 1:

for i := 1 **step** 1 **until** sin (pi × a) **do** $b[i]$:= 0 ;

Optimized version:

n := sin (pi × a) ;

for i := 1 **step** 1 **until** n **do** $b[i]$:= 0 ;

Example 2:

for i := 1 **step** 2 × i **until** 101 **do** $b[i]$:= 0 ;

No optimization is possible here because the expression 2 × i changes its value during the execution of the for statement. It should be noted that this "optimization" is "translator dependent." In example 1, most translators will produce a better machine program from the second or optimized version of the computing process. Very efficient translators might give better programs from the first version.

value of loop variable

5. **Value of Loop Variable on Exit from For Statement.** – The execution of a for statement may be terminated in two ways: (a) exhaustion of the for list; (b) execution, inside the for statement, of a go to statement leading out of the for statement. In case (a) the value of the loop variable is not defined. In case (b) it is defined to be the value of the loop variable at the time of execution of the go to statement.

go to into for statement

6. **Go To Statement Leading into a For Statement.** – Such a go to statement is illegitimate.

DECLARATIONS AND BLOCKS

Type and Array Declarations

For every simple variable which appears in an ALGOL program there must be a type declaration, and for every array there must be an array declaration. A *type declaration* determines the range of values which a variable may assume. There are the three types: **real, integer,** and **boolean.**

types

A **boolean** variable may assume only the values **true** and **false**. An **integer** variable may assume those integer values which are representable in a particular machine (ALGOL does not attempt a standardization in this respect). A **real** variable may assume every real value representable in a particular machine (ALGOL does not attempt a standardization, but it is tacitly understood in all ALGOL programs written so far that the numbers are represented in floating-point form, with a decimal exponent range of about −50 to +50 or an equivalent binary exponent range). A type declaration has one of the following three forms:

boolean $V_1, V_2, \ldots, V_n$;

integer $V_1, V_2, \ldots, V_n$;

real $V_1, V_2, \ldots, V_n$;

mixed expressions

It is permissible to mix variables or numbers of types integer and real in arithmetic expressions. The result of a mixed operation is of type real. The result of a division of two integers is of type real. The integer division operation ÷, however, is only defined if a and b are of type integers. If a/b is positive, then $a \div b$ = entier (a/b). Otherwise $a \div b = -$ entier $(-a/b)$. Thus

integer division

$$a \div b = (-a) \div (-b) = -(a \div (-b)) = -((-a) \div b) .$$

no Boolean values in arithmetic operations, and conversely

It is not permissible to use a Boolean variable or constant as an operand of an arithmetic operation. No number, real or integer, is associated with the truth values **true** and **false**. Conversely, no truth value is associated with numbers.

mixed assignment statements

The assignment of a noninteger value to an integer variable is always understood in the sense of proper round-off; that is, the value assigned to the integer variable will be the integer closest to the noninteger value. If a is of type integer the following pairs of statements are equivalent:

$a := 1.4$	and	$a := 1$
$a := 1.5$	and	$a := 2$
$a := 1.7$	and	$a := 2$
$a := -1.1$	and	$a := -1$
$a := -1.5$	and	$a := -1$

(Note: There is a slight distinction between an integer number of type real and an integer number of type integer with respect to the operation ÷. The operation $a \div b$ is not defined if one of the variables a or b is of type real, even if the values of a and b are both integer at the time when the operation $a \div b$ is to be executed.)

The type of numbers is inherent in the string of characters by which they are represented. Every number made up from the symbols +, −, 0, 1, . . . , 9 is of type integer. Every other number is of type real, although its value may be integer, for example, the numbers 3.0 and 0.4 ₁₀1. The reader is referred to the examples and counterexamples for numbers in Table 1. The distinction between numbers of type integer and type real matters only in connection with the operation ÷. Otherwise it is immaterial.

The assignment of a truth value to an arithmetic variable or the assignment of an arithmetic value to a Boolean variable is not defined.

subscript bounds in array declarations

An *array declaration* gives bounds for the values which the subscripts of an array may assume. In addition, the type of the array may be given in an array declaration. If no type is given, the array is understood to be real. (Note: No such declaration is implied for simple variables; their type must be declared.) All elements of an array are of the same type. The following examples make the structure of array declarations sufficiently clear.

type of arrays

array a, $b[1 : 10]$, c, d, $e[1 : 14, 6 : 9]$, $f[-1 : +2]$;

This declaration means: The arrays a and b are one-dimensional arrays, and the subscript ranges between 1 and 10. The arrays c, d, e are two-dimensional arrays, with the first subscript ranging from 1 to 14 and the second from 6 to 9, etc. All these arrays are reul.

The lower bound for a subscript must be written before the upper. Thus **array** $a[4 : 1]$ is not a valid array declaration. References to a subscripted variable outside the range of the subscript bounds are invalid. A program with the given array declaration which uses the subscripted variables $a[0]$ or $c[-1, 18]$ would be incorrect. Other examples for array declarations are:

lower bound comes first

integer array $d[1 : 14]$;

real array $g[1 : 14, 8 : 9, -3 : -1]$, h, $f[1 : 2]$

A program may be constructed from one or several *blocks*. A block is, roughly speaking, a compound statement that contains declarations about the variables which are "local" to the block. Such local variables must not be used by a statement not contained in the block. The declarations for a block have to follow immediately the **begin** which indicates the beginning for the block; the statements, of which the block is composed, follow the declarations, and the block is terminated by the **end** which matches the block-**begin**.

position of declarations

local quantities

A block is also considered to be a statement, and thus may be a constituent statement of another block.

Two blocks B_1 and B_2 may be related to each other in three different ways:

1. B_1 is a subblock of B_2,
2. B_2 is a subblock of B_1,
3. B_1 and B_2 are independent blocks.

For case 1 to be true, B_1 either is a constituent statement of B_2, or B_1 is a subblock of a subblock of B_2. For case 3 to be true, B_1 and B_2 are either constituent statements of a block B, or they are subblocks of different constituent statements of B.

Example:

```
L : begin   real   a, b, c ;
                .
                .
                .
M :             begin   integer   a ;
                        .
                        .
                        .
N :                     begin   real   a, b ;
                        .
                        .
                        .
                        end
O :                     begin   real   c, d ;
                        .
                        .
                        .
                        end
                end
P :             begin   real   d ;
                .
                .
                .
                end
                .
                .
                .
      end
```

In the example given, the blocks labeled *M, N, O, P* are subblocks of *L*; the blocks labeled *N, O* are subblocks of *M*. The blocks *M, N, O* are independent of *P*, and *P* is independent of *M, N, O*. The block *N* is independent of *O* and vice versa.

Those statements of block *L* which are not contained in one of its subblocks may refer to the variables *a, b, c*. The statements of block *M* which are not contained in one of its subblocks may refer to the variables *a, b, c*. Notice, however, that the *a* of block *M* is different from the *a* of block *L*, whereas the variables *b* and *c* used in *M* are the same as those used in *L*. Or to put it another way: A variable which is declared in a block B_1 is valid for all those statements and subblocks of B_1 which do not contain a declaration for a quantity with the same name. It is not valid in any of those blocks which are independent of B_1 and, outside B_1, in those blocks of which B_1 is a subblock.

There are, however, the possibilities, which are to be differentiated, that a variable of a block B_1 becomes invalid because control is transferred to a subblock B_2, where a variable with the same name is declared, and that a variable becomes invalid as a result of leaving the block B_1 where it is declared. In the first case, the value of the invalid variable is retained and available after exit from B_2. In the second case, the value of the invalid variable is lost, and is not available on re-entering B_2.

There are consequently three possibilities for the "state" of variables:

1. valid,
2. invalid, value defined,
3. invalid, value not defined.

Table 2 gives the state of each variable in the program on p 23 as a function of the block.

Table 2. State of Variables as Functions of Blocks

Variable	Block L	Block M	Block N	Block O	Block P
a of block L	Valid	Not valid, defined	Not valid, defined	Not valid, defined	Valid
b of block L	Valid	Valid	Not valid, defined	Valid	Valid
c of block L	Valid	Valid	Valid	Not valid, defined	Valid
a of block M	Not valid, not defined	Valid	Not valid, defined	Valid	Not valid, not defined
a of block N	Not valid, not defined	Not valid, not defined	Valid	Not valid, not defined	Not valid, not defined
b of block N	Not valid, not defined	Not valid, not defined	Valid	Not valid, not defined	Not valid, not defined
c of block O	Not valid, not defined	Not valid, not defined	Not valid, not defined	Valid	Not valid, not defined
d of block O	Not valid, not defined	Not valid, not defined	Not valid, not defined	Valid	Not valid, not defined
d of block P	Not valid, not defined	Not valid, not defined	Not valid, not defined	Not valid, not defined	Valid

A quantity which is valid in a block B but which is not declared in B is called a *global* quantity in B.

global quantities

Storage Allocation in a Program with Block Structure

Here is one possible and easy way to allocate storage to the variables in a program with block structure: Each variable declared is allocated a unique storage location. In the program given above this would amount to reserving nine different storage locations for the nine different variables. This is wasteful, since it is possible to have only six different locations in the following way:

Loc 1	:	a of block L
Loc 2	:	b of block L
Loc 3	:	c of block L
Loc 4	:	a of block M, and d of block P
Loc 5	:	a of block N, and c of block O
Loc 6	:	b of block N, and d of block O

This saving of storage space is not important in the case of simple variables; it may be decisive in the case of arrays. An ALGOL translator can take advantage of this possibility given by the block structure. The block structure gives among other things most of the features of the "Common" and "Equivalent" statements of FORTRAN.

The discussion on "valid" and "defined" quantities just given for simple variables also applies to arrays. The rather interesting feature here lies in the fact that the size of a local array may depend on quantities computed outside the array.

subscript expressions for arrays

The subscript bounds in array declarations are arithmetic expressions. They may contain variables and procedures (see below) which are global to the block in which the array is declared. They must not contain variables and procedures which are local to this block. If one wants to make efficient use of this feature, one must allow for "dynamic storage allocation" of the elements of an array. This means that storage space for the elements of an array is allotted at execution time, more specifically, at the time when control enters the block where the array is declared.

Local Labels

Labels are, in the same way as variables and arrays, denoted by "free names." There are no explicit declarations for labels. The fact that a name denotes a label is implied by the way in which the name is used. A name or an unsigned integer which appears immediately in front of a statement, separated from the statement by a colon, is a label, and such an appearance of a name may be considered as a "label declaration." There is of course only one declaration for each label in each block. If L is a label "declared" somewhere, then the statement "**go to** L" may be called a statement "using" that label.

label declaration

All labels declared in a block are local to the block, and in this sense the notions of validity apply to labels in the same way as to variables. From this statement, it follows immediately that one cannot jump to a label inside a block by means of a go to statement which is outside the block. Also, if in block B_1 a label L is declared, and if B_2 is a subblock of B_1 where again L is declared, then any statement **go to** L inside B_2 refers to the label L in B_2. Any statement **go to** L in B_1 refers to label L in B_1.

jump into a block not permissible

A compound statement which does not contain any explicit declarations is not a block, and labels of compound statements which are not blocks are not "local" to that compound statement.

labels of compound statements which are not blocks are not local

Switches

Assume that in an ALGOL program one has to write a statement which transfers control to one of five different labels L, P, Q, L_2, L_1, depending on whether a variable i is equal to 1, 2, . . . , 5. One can write the statement:

example for use of a switch

```
if $i = 1$ then go to $L$ else if $i = 2$ then go to $P$
   else if $i = 3$ then go to $Q$ else if $i = 4$ then go to $L_2$
   else go to $L_1$.
```

One can greatly simplify this statement by combining the five labels L, P, Q, L_2, L_1 into a *switch* by means of a switch declaration:

switch $s := L, P, Q, L_2, L_1$,

and replacing the lengthy conditional statement by:

go to $s[i]$.

The switch declaration in this example declares the label L to be the first element of the switch with name s, label P to be second element of this switch, etc. Reference to elements of a switch is made in a way analogous to referencing elements of one-dimensional arrays:[8] The name of a switch followed by a left bracket followed by an expression **E** followed by a right bracket designates the *k*th label of the switch, when *k* is the integer closest to the value of expression **E**.

designational expression

A *designational expression* is defined to be

(*a*) a label,

(*b*) a structure of the form **S[E]**,

where **S** is the name of a switch.

Designational expressions are primarily used in go to statements (see the example above). They may, however, also be used in defining a switch.

switch declaration

If $\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_n$ are designational expressions, and **I** is an identifier, then

$$\textbf{switch } \mathbf{I} := \mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_n$$

is a *switch declaration.*

The statement **go to** $\mathbf{I}[\mathbf{E}_1]$ transfers control to the *k*th designational expression of switch **I**, when *k* is the integer nearest to $\mathbf{E}_1$. If this designational expression is a label *L*, the statement is equivalent to **go to** *L*. If it is of the form $s[\mathbf{E}_2]$, the statement is equivalent to **go to** $s[\mathbf{E}_2]$; that is, it refers to the *j*th designational expression in the declaration for switch *s*, where *j* is the nearest to $\mathbf{E}_2$. This process of referring to other switches within a switch may be repeated an arbitrary number of times. Such recursive switches, however, are rarely used.

conditional designational expression

It might be mentioned that one can form "conditional designational expressions." An example may suffice:

$$\textbf{go to if } a = b \textbf{ then } L \textbf{ else if } a \leq b \textbf{ then } P \textbf{ else } Q$$

The use of conditional designational expressions can and should be avoided. The statement

$$\textbf{if } a = b \textbf{ then go to } L \textbf{ else go to } M$$

is better "ALGOL style" than the statement

$$\textbf{go to if } a = b \textbf{ then } L \textbf{ else } M\ .$$

A switch declaration may, within a program, occur in any place in which type and array declarations may occur. The variables and labels

[8] Switches may be considered as a kind of one-dimensional array; the "values" of elements of this array are not numbers but labels.

used in switch declarations must be valid in the block in which the switch is declared. A switch declared in a block is, of course, local to that block in the sense described above in connection with simple variables.

local switches

PROCEDURES

General Discussion

Procedures serve, in ALGOL, the same purpose which subroutines serve in ordinary machine coding.

A piece of ALGOL program, which is used in several places of a program or in several programs, with possibly different *parameters*, may be declared a procedure by preceding it with a *procedure heading*. A procedure heading may in some simple cases have the form:

procedure heading, simple form

$$\textbf{procedure I}(\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n);$$

where **I** is an identifier (the "name" of the procedure), and $\mathbf{P}_1, \mathbf{P}_2, \ldots,$ $\mathbf{P}_n$ are identifiers, which denote the *formal parameters* of the procedure.

formal parameter

More elaborate forms of the procedure heading are considered below.

The piece of program associated with the name **I** is called a *procedure body*. The procedure body is an ALGOL statement. Normally it is a block, since most procedures use local quantities declared in the procedure body. The procedure heading and procedure body together form the *procedure declaration*.

procedure body

procedure declaration

Execution of the procedure body is initiated by a *procedure call*. A procedure call is a statement. The procedure with name **I** is called by the statement:

procedure call is a statement

$$\mathbf{I}(\mathbf{AP}_1, \mathbf{AP}_2, \ldots, \mathbf{AP}_n),$$

where $\mathbf{AP}_1, \mathbf{AP}_2, \ldots, \mathbf{AP}_n$ denote *actual parameters* of the procedure call.

An actual parameter may be:

actual parameters

1. an expression (arithmetic, Boolean, designational),
2. an identifier denoting a procedure, a switch, or an array,
3. a formal parameter, if the call appears in the body of a procedure.

Executing a procedure statement means execution of the procedure body after the following changes have been made:

1. The formal parameters of the procedure are replaced, in the sense of copying, by the corresponding actual parameters of the procedure call, after enclosing these in parentheses whenever this is syntactically possible.
2. The names of local quantities are changed so that they are different from all names appearing in the actual parameters.

copy rule

An addition to this rule is necessary if some of the parameters are called "by value" (see below).

for an addition to the copy rule in case of value parameters see below

A Simple ALGOL Program Containing a Procedure Declaration

Read in a sequence of number quadruples a, b, c, d. Compute the area of all triangles which can be formed with sides equal to any three of a, b, c, or d, and punch these. The program uses a procedure which computes, for three numbers x, y, z, the area a of a triangle with sides of these lengths according to the formula $a = \sqrt{s(s-x)(s-y)(s-z)}$, where $s = (x + y + z)/2$. The result a is punched. The program also tests whether a triangle can be formed of sides with lengths x, y, z:

```
                                                          Line
    real a, b, c, d ;                                        1
    procedure triangle area (x, y, z) ;                      2
             begin real s, a ;                               3
                  s := 0.5 × (x + y + z) ;                   4
                  if s ≥ x ∧ s ≥ y ∧ s ≥ z then              5
                    begin a := sqrt(s × (s−x) × (s−y) × (s−z)) ;   6
                         punch a                             7
                    end                                      8
                    else punch −1                            9
             end triangle area ;                            10

L : read a, b, c, d ;                                       11
    carriage return ;                                       12
    triangle area (a, b, c) ;                               13
    triangle area (a, b, d) ;                               14
    triangle area (a, c, d) ;                               15
    triangle area (b, c, d) ;                               16

    go to L ;                                               17
```

Notes on the triangle program:

1. Line 1: These are declarations concerning the variables used in the program.
2. Line 2: The identifier following the procedure is the name of the procedure. Its parameters are x, y, z. Consistent replacement of all the identifiers used as formal parameters does not affect the meaning of the procedure or of the program in which it is declared. We could even change x, y, z into identifiers which are equal to some identifiers used elsewhere in the program; for example, we could replace x by b. We could of course not replace x, y or z by s or a.

procedure names are arbitrary identifiers

3. Line 3: Declarations of variables local to the procedure, that is, local in the sense described in "Declarations and Blocks."
4. Lines 4 to 10: The statements of the procedure body.
5. Line 13: This procedure statement is equivalent to the execution of the procedure body after substituting a, b, c for x, y, z, and after changing the local quantity a in the procedure body into some other identifier.

The statement which results from the copying process described above must be a valid ALGOL statement. There is no other rule or restriction in forming the procedure body. For some difficulties which may result in some cases, see "Recursive Procedures" and "Own Variables."

Second Example of a Program with Procedures

Form:

$$\sum_{i=1}^{10} i^2 , \quad \sum_{j=5}^{10} \sin j , \quad \sum_{j=6}^{10} p[j] ,$$

where the $p[j]$ are to be read in, and punch these three sums:

```
begin real a, b; integer i, j, k; array p[0:10]
      procedure sum (x, y, f, m, s) ;
            begin s := 0 ;
                  for m := x step 1 until y do s := s + f
            end ;
      sum (1, 10, i ↑ 2, i, k) ;
      sum (5, 10, sin j, j, a) ;
      for i := 6 step 1 until 10 do read p[i] ;
      sum (6, 10, p[j], j, b) ;
      punch k, a, b ;
end
```

The copying process transforms this program into:

```
begin real a, b; integer i, j, k ; array p[0:10] ;
      begin k := 0 ;
            for i := 1 step 1 until 10 do k := k + i ↑ 2
      end ;
      begin a := 0 ;
            for j := 5 step 1 until 10 do a := a + sin(j)
      end ;
      for i := 6 step 1 until 10 do read p[i] ;
      begin b := 0 ;
            for j := 6 step 1 until 10 do b := b + p[j]
      end ;
      punch k, a, b ;
end
```

Note that the third parameter of the procedure sum is replaced by an expression. However, it is not the value of this expression which is transmitted to the procedure; rather, the *rule for computing an expression* is transmitted and replaces the formal parameter. This device is at the same time powerful for ease of expressing algorithms and troublesome for compiler builders. Techniques to handle this situation are discussed in some of the papers in the January 1961 issue of the *Communications of the ACM*.

General Discussion, Continued

It is, of course, not the intention of the copy rule that, before translating an ALGOL program into machine code, all procedure calls are replaced by the bodies of the called procedures. The copy rule is a simple way of telling what a procedure call means. In actual translation of a program one tries to have only one copy of the (translated) procedure in the machine at the time of execution and transfer control to this piece of program for each procedure call. The question of how to do this is interesting, but it will not be discussed here.

position of procedure declaration

A procedure declaration may be placed where a type, switch, or array declaration is permissible. A procedure is local to the block where it is declared in the sense described for simple variables. This means that a procedure cannot be called by a statement outside the

local properties of procedures

block in the heading of which the procedure is declared. And it cannot be called by a statement inside a block which contains a declaration for a variable, array, switch, or procedure which has the same name as the procedure in question.

It also means that one may have two procedures with the same name in different blocks of a program (e.g., two procedures with the name "triangle area" which use different formulas for computing that area).

global quantities in procedures

Identifiers used inside procedures are either formal parameters, local quantities, or names of quantities defined outside the procedure. These latter quantities are global to the procedure. The following example[9] shows such a situation:

```
                                              Line
begin real a, b ;                              1
      procedure P(x, y, z) ;                   2
      begin real r, s ;                        3
            a := x + y                         4
      end ;                                    5
      a := 6; p(1, 2, 3); punch a              6
end                                            7
```

The variable a appearing in the body of procedure P is not a parameter and not a local quantity, so it is a global quantity. After execution of the procedure statement in line 6, the value of a will be 3. Had there been a declaration for a (e.g., **real** a) in the body of P, the value of a after the procedure call would still be 6.

There are some cases where the interpretation of the copy rule is dubious. Consider the following example:

```
                                       Line
begin real a, b ;                       1
      procedure P(x, y) ;               2
      begin real r ;                    3
            a := x + y                  4
      end ;                             5
```

[9]This "program" does not make much sense, but it serves our purpose of illustrating the properties of global quantities in procedures.

```
          begin real a ;                    6
                a := 1 ;                    7
                P(1, 2) ;                   8
                punch a ;                   9
          end                               10
      end                                   11
```

The question here is the interpretation of the quantity *a* in the procedure call of line 8. If the copy rule is taken literally, the procedure statement in line 8 is equivalent to

```
begin real r ;
      a := 1 + 2
end ;
```

This would of course mean that the value punched in line 9 is 3. One might suspect, however, that all translators now under construction – except those which actually make one copy for each procedure call – will punch the number 1 in line 9 of the program. This means that a procedure takes its global quantities from the block where it is defined and not from the block where it is called.

Specifications

A consequence of the use of the copy rule in the definition of ALGOL procedures is that it is very difficult to translate a procedure declaration independent of the procedure calls.[10] Usually, a translating program is guided by the declarations for the various entities occurring in the program. There are, however, no declarations for the parameters. Declarations for parameters are "inherited" from the declarations for the actual parameters used in calling the procedure.

The following example shows a difficulty which arises from this:[11]

```
begin real b, c, d, x ;
      boolean A, B ;
      procedure P(p, q, r, s, v) ;
      if if p then q else r ≤ s then v := 1 else v := 2 ;
      P(A, B, c, d, x) ;
      P(A, b, c, d, x)
end
  .
  .
  .
```

[10]For a thorough discussion of the problems involved and a possible (though not efficient) solution, see (4).

[11]Discussed in similar form by H. Rutishauser in *ALGOL-Bulletin*, No. 10, p 11, edited by Regnecentralen, Copenhagen-Valby, Denmark, 1960.

According to the copy rule, the first procedure call is equivalent to:

if if A **then** B **else** $c \leq d$ **then** $x := 1$ **else** $x := 2$

Using parentheses this may be written

if (**if** A **then** B **else** $(c \leq d)$) **then** $x := 1$ **else** $x := 2$;

The second procedure call is equivalent to:

if ((**if** A **then** b **else** $c) \leq d$) **then** $x := 1$ **else** $x := 2$

Thus, the scope of the second **if** in the procedure body depends on the parameters. This means that it is not possible to translate the procedure declaration for P without knowing which actual parameters are used in calls of this procedure.

This example also suggests that one should use parentheses to indicate the structure of conditional expressions, even though the parentheses may not be necessary. Such redundant parentheses help in writing, reading, and translating such expressions.

There are other, more difficult cases where a procedure can only be translated after examination of parameters used in calling the procedure. Many of these problems can be overcome by *specifications*. A specification gives information about the formal parameters of a procedure declaration. A specification is a *specifier*, followed by a list of identifiers. There are the following specifiers:

specification

specifiers

label
switch
string
real, integer, boolean
procedure, real procedure,[12] **integer procedure,**[12] **boolean procedure**[12]
array, real array, integer array, boolean array

Specifications have to be written immediately preceding the procedure body. They are separated by semicolons from the parameter list, the procedure body, and each other. For an example of a procedure declaration with specifications, see procedure *Bessel*, p 35.

A specification restricts in an almost obvious way the actual parameters which may be substituted for a specified formal parameter. Thus, a parameter which is specified as **real** may only be replaced by arithmetic expressions. Or a parameter which is specified as a label may only be replaced by labels, or by a designational expression. In the example on **p** 33, if the dubious parameter q is specified as **boolean**, the second call of the procedure would be illegal, and the procedure P

[12]Concerning these specifiers compare p 38.

could be easily translated in such a way that all legal procedure calls would be executed correctly.

Value Parameters

Consider the example on p 3, for computing approximations to the values of Bessel functions. We will write this program in procedure form, making a few changes. The parameters of the procedure are x, n, *sum*, and the procedure computes an approximation to $I_n(x)$ and assigns that value to the variable sum. We omit from the example on p 3 the loop which computes $I_n(x)$ for different values of x, and we omit the read and punch statements. We then get the following procedure:

```
        procedure Bessel (x, n, sum); real x, sum; integer n ;
        begin integer nfac, i, s; real denom ;
              nfac := 1 ;
              for i := 2 step 1 until n do nfac := nfac × i ;
              if x = 0 ∧ n = 0 then begin sum := 1; go to P end ;
    L :       denom := nfac ;
              sum := 0 ;
              for s := 0 step 1 until 10 do
                      begin sum := sum + (x/2) ↑ (n + 2 × s)/denom ;
                            denom := denom × (s + 1) × (s + 1 + n)
                      end ;
    P : end Bessel
```

A possible call of this procedure would be:

$$Bessel\ (n/(i + 4 \times x),\ 4,\ f)$$

According to the copy rule this is equivalent to executing the procedure body, replacing x by $n/(i + 4 \times x)$, *sum* by f, and n by 4, and substituting a new name for i to make it different from the i in the first parameter of the procedure statement.

During execution of the procedure the value of $n/(i + 4 \times x)$ would be computed 12 times, always resulting in the same value, since neither n nor i changes inside the procedure. An intelligent translator could find out that $n/(i + 4 \times x)$ need be computed only once and would program accordingly. However, in order to simplify efficient translation, a formal parameter may be declared a *value parameter* in the procedure heading. This means that whenever such a parameter is replaced by an expression in a procedure call, the value of that expression is obtained

value parameters

and assigned to that parameter before execution of the procedure. This formal parameter is treated as a local quantity in the procedure body, and the name of this parameter must possibly be changed in the same fashion as the names of the other local quantities.

execution of a procedure with value parameters

A value parameter must be specified in the procedure heading. In the example above, if x is a value parameter, the procedure heading would look as follows:

value parameters must be specified

procedure *Bessel* (x, n, *sum*) ; **value** x ; **real** x ;

The value declarations must precede all of the specifications, even the specifications for the nonvalue parameters.

value part first, then specifications

With the extended rule for execution of a procedure statement, the statement "*Bessel* ($n/(i + 4 \times x)$, 4, f)" is equivalent to the following block:

```
                                      Notes
begin integer nfac, istar, s ;        xstar and istar are the names
                                      substituted for x and i
   real denom, xstar ;
   xstar := n/(i + 4 × x) ;           assignment of the value to the
                                      value declared parameter
   nfac := 1 ;
   for istar := 2 step 1 until 4 do   n is replaced by 4
      nfac := nfac × istar ;
   if xstar = 0 ∧ 4 = 0 then
     begin f := 1 ; go to P end ;     sum is replaced by f
   .
   .
   .
   etc.
   .
   .
   .
P :
end
```

The reasoning given above for introduction of value parameters seemed to imply that it affects only the efficiency of the procedure statements.

addition of value part changes the meaning of a procedure

But consider the example:

```
procedure A (x, y) ;
begin
    x := . . .
    y := . . .
end
```

The procedure call $A\ (a + b,\ 3)$ is invalid because it would involve execution of the "statements":

$$a + b := \ldots$$
$$3 := \ldots$$

If one adds the value part, as follows:

```
procedure A (x, y) ; value x, y ; real x, y ;
begin
    x := . . .
    y := . . .
end ,
```

the above-mentioned procedure call will be executed as:

```
begin real x, y ;
    x := a + b ; y := 3
    .
    .
    .
    x := . . .
    y := . . .
end ;
```

This is a valid ALGOL statement, and thus the procedure call is valid. The addition of the **value** declaration thus affects the class of actual parameters which may be substituted for a formal parameter.

Functions

Let it be required to compute the expression

$$E = \sum_{k=1}^{5} a_k \cdot I_k(k \cdot x) ,$$

where the a_k are stored in an array. By using the procedure *Bessel*, E can be computed by the following piece of program:

```
E := 0 ;
for k := 1 step 1 until 5 do
   begin Bessel (k × x, k, f) ;
         E := E + f × a[k]
   end
```

functions

It is desirable to use the "result" *f* of the procedure *Bessel* immediately in an arithmetic expression. For this purpose, *functions* have been introduced as a special kind of procedure, namely, those with one particularly interesting "result."

declaration of functions

type of results of functions

In the declaration of a procedure which is a function, this "result" is denoted by a variable which has the same name as the procedure. The type of this variable must be declared by placing the type immediately in front of the word **procedure**. As an example we will write procedure *Bessel* as a function. We use the name *Besselfunction* for this procedure in order to distinguish it from the procedure *Bessel* on p 35:

```
        real procedure Besselfunction (x, n);
        value x, n; integer n; real x;
        begin integer nfac, i, s; real denom, b;
           nfac := 1;
           for i := 2 step 1 until n do nfac := nfac × i;
           if x = 0 ∧ n = 0 then begin b := 1; go to P end;
L  :       denom := nfac; b := 0;
           for s := 0 step 1 until 10 do
              begin b := b + (x/2) ↑ (n + 2 × s)/denom;
                    denom := denom × (s + 1) × (s + 1 + n)
              end;
P  :       Besselfunction := b
        end
```

the name of a function and the name of its result are identical

In the body of the **real procedure** *Besselfunction* the name *Besselfunction* is used to designate the result. Note that "*Besselfunction*" is not a local variable of the procedure body. There is no declaration for such a variable. Also, the result may not be used in an arithmetic expression inside the procedure. This is necessary if one wants to

result may not be used in expressions inside procedure avoid confusion with recursive procedures; see p 40. This restriction accounts for a minor deviation of the procedure *Besselfunction* from the procedure *Bessel*: The quantity sum of the latter procedure is replaced by the quantities *b* or *Besselfunction*, because we no longer can use the same name for an intermediate quantity and for the result.

With this procedure *Besselfunction* the computation of the sum

$$\sum_{k=1}^{5} a_k \cdot I_k(k \cdot x)$$

can be described in the following way:

```
    $E := 0$ ;
    for $k$ := 1 step 1 until 5 do $E := E + Besselfunction\ (k \times x,\ k)$ ;
```

call of functions

In general, a procedure which is a function is called by writing its name in an expression and placing after the name a list of parameters enclosed in parentheses. Some difficulties arise if a function changes the values of global parameters, and the exact interpretation of this case was the subject of much discussion since the appearance of the ALGOL 60 report. Since this difficulty, however, arises only in very rare cases and can always be avoided by simple means, we will not discuss this topic.

For an example of Boolean functions see p 44.

Boolean function, see p 44

Procedures in Machine Code

machine code in ALGOL programs

Certain operations or algorithms cannot be expressed efficiently in ALGOL. In this class belongs manipulation of quantities which occupy only a few bits of a computer word, or the double length accummulation of a sum of products. Procedures to handle such computations can be written in *machine code*. ALGOL does not specify anything about the form in which such machine code procedures should be written. However, it is part of ALGOL that machine code procedures can be called by an ALGOL procedure statement, *with no restriction on the type of parameters used in the procedure call.* Therefore the writing of a machine code procedure must take into account the way in which an ALGOL procedure statement is translated.

Another area of application of machine code procedures is manipulation of auxiliary equipment such as drums, files, tapes, etc. Input and output of information can be incorporated in an ALGOL translator by means of machine code procedures.

Recursive Procedures

A procedure P which calls, in its body, itself, or which calls another procedure P_1 which calls P, is said to be a *recursive procedure*. Consider the following example:

```
real procedure factorial (n);
      if n = 1 then factorial := 1 else factorial := n × factorial (n − 1);
```

This is a recursive procedure, because the execution of this procedure, for example, in case $n = 2$ requires the execution of this same procedure for $n = 1$. It is not a very good program for the computation of the factorial, since it requires n procedure calls. Even on computers with fast subroutine jump facilities, it will probably use more time for jumps to the subroutine and back than it does for the actual computation. The factorial should be programmed with a loop such as:

```
real procedure factorial (n) ; value n ; integer n ;
    begin integer i , f ;
          f := 1 ;
          for i := 1 step 1 until n do f := f × i ;
          factorial := f
    end ;
```

The last program performs an "iterative" computation of the factorial, as contrasted with the "recursive" computation given before. A large part of many programming efforts consists in reducing recursive processes to iterative processes. In some areas, however, this reduction is either not possible or very cumbersome, and in such cases recursive procedures should be used. One area for the application of recursive procedures is translator construction (5).

Recursive Procedures and Copy Rule

The copy rule (p 29) allows us to eliminate procedure declarations and procedure calls from a program by actually replacing each procedure call by the body of the called procedure, with the changes required by the copy rule. Although, as has been pointed out before, it is not desirable to actually make this copy in a translator, the copy rule is a simple way of describing the meaning of a procedure call. Evidently the copy rule does not work for recursive procedures: Every copy produced would call for another copy, and the copying would go on indefinitely.

The copy rule can be modified in such a way that a copy of the called procedure is produced only after a call of the procedure has been encountered. This interpretation of the copy rule still leaves the following question, which is not answered by the ALGOL 60 report: Will the names of local quantities be changed in the same way in the different copies of a single procedure, or will these changes be made independently. In the first case, every level of a recursive procedure acts on the same set of local quantities. In the second case, every level has its own local quantities. At present the second interpretation seems to be most commonly accepted. It seems that in those cases where recursive procedures are really important both kinds of quantities are desirable. Most translators presently under construction will not handle recursive procedures, so that the question raised above is at present not of great practical importance.

Own Variables

The value of a variable is lost after exit from the block in which this variable is declared. There are some cases where this is undesirable, and ALGOL provides for a special class of local variables, the so-called *own* variables, which retain their identity throughout the program. A simple variable or an array is declared own by preceding the corresponding declaration with the symbol **own**. Example:

own variables and arrays

```
        own real x, y ;
        own integer array a[1 : 10] , b, c[4 : 17] ;
```

For a precise interpretation of these declarations, consider the following example:

the behavior of the own variable *y* and the own array *b* of this block will be discussed below

```
          .
          .
          .
          n := 15 ;
    R  :  go to L ;
    L  :  begin real x ; own real y ;
                 array a[1 : n] ; own real array b[1 : n] ;
                 .
                 .
                 .
                 x := 4 ; y := 6
          end
          .
          .
          .
    M  :  begin integer a, b ;
                 array x, y[1 : 5] ;
                 .
                 .
                 .
          end ;
          if n = 15 then n := 20 else n := 15;
    P  :  go to L ;
```

During execution of this program, storage space is reserved for the variable y of the block labeled L. Outside block L no reference can be made to this variable. In the block labeled M the identifier y is used to denote an array. This use of y does not, of course, interfere with the variable y of block L. When block L is left in the normal way, that is, after executing the statements "$y := 6$; $x := 4$," the value of the variable y will still be 6 on re-entry to the block. The value of x, which is not own, will be lost after block L is left, and the value of x is undefined after re-entry to block L. The location of x could, for instance, be used by the variable a of block M.

If the block L is entered from the statement labeled R, the value of n is 15. On the first entry to block L, locations for $u[1]$ to $u[15]$ and $b[1]$ to $b[15]$ will be reserved. On re-entry to L from statement P, n will be 20. At this stage, locations are still reserved for $b[1]$ to $b[15]$ (not necessarily the same ones as on the previous exit), and these locations contain the values which $b[1]$ to $b[15]$ had on the previous exit. Before the first statement of block L is executed, locations for $a[1]$ to $a[20]$ and $b[16]$ to $b[20]$ will be reserved. The values of these variables are, of course, not defined. Before the next entry to block L from statement R, the value of n will be reset to 15. After entry to block L and before execution of the first statement of this block, storage reservation for array b will be restricted to $b[1]$ to $b[15]$. The values of $b[16]$ to $b[20]$ will be lost. They will not be recovered after the next entry to block L, even if n is reset to 20.

A local variable must appear on the left side of an assignment statement or in a **read** statement before its value can be used in an arithmetic expression. This is also true for own variables of a block when the block is executed for the first time. On subsequent entries to the block the assignment of a value to an own variable may be bypassed. As a matter of fact, if the assignment of a value to an own variable is not sometimes bypassed there is no sense in making the variable own, because its value, although available at the beginning of a block, will be recomputed before it is used.

Own Variables in Procedures

own variables in procedures

If the block which constitutes a procedure body contains own variables, the question mentioned under recursive procedures, p 40, comes up again: Are the changes of names for own quantities which are made in the different copies corresponding to different procedure calls identical, or are they made independently? In case of independent changes, each procedure call would have its own "own variables" which are not affected by other calls of the same procedure. In the other case, all calls of the same procedure act on the same set of own variables. It should be noted that own variables are very awkward to use, and are very awkward to handle by a translator when the first interpretation is made.

The question of identical vs independent changes does not arise in case of local, nonown quantities in nonrecursive procedures. Their values are not defined after exit from the procedure, so that there can be no relation between the local quantities used in different procedure calls.

Special Parameter Delimiters

for special parameter delimiters, see p 48

EXAMPLES OF ALGOL PROCEDURES

In the remainder of this report we will give some ALGOL procedures for internal sorting. It is sometimes contended that ALGOL, though it may be adequate for expressing procedures which are mainly numerical, is unsuited for nonnumerical algorithms such as sorting. We chose our examples from the general area of sorting in order to show that these "logical," rather than numerical, procedures can be adequately expressed in ALGOL.

Program for Binary Search

The following procedure assumes that the elements of an array *a* are arranged in descending order, that is, $a[1] \geqq a[2] \geqq a[3], \dots$. Given a number *b* and a subscript *l* such that

$$a[1] \geqq b \geqq a[l]$$

the program determines in $[\log_2 l]$ comparisons a subscript *p* for which

$$a[p] \geqq b > a[p+1] \quad \text{or} \quad a[p] = b = a[p+1] .$$

```
procedure binary search (a, b, l, p) ; value l, b ; integer l, p ;
          real b ; real array a ;
begin integer p1 ;
          p := 1 ;
test for end :   if l - p = 1 then go to M ;
          p1 := (l + p) ÷ 2 ;
          if a[p1] < b then l := p1 else p := p1 ;
          go to test for end ;
M :
end ;
```

By using a for statement with an "*E* **while** *B*" element, this procedure can be expressed a little more elegantly, though perhaps this is not so easily understood by a reader not familiar with this type of for statement:

```
    p := 1 ;
    for p1 := (l + p) ÷ 2 while l − p ≠ 1 do
        begin if a[p1] < b then l := p1 else p := p1 end ;
```

The Binary Search Program with a Boolean Function as a Parameter

The *binary search* procedure above works only if the elements $a[1]$, $a[2]$, ... are arranged in descending order. If one needs, as part of a larger algorithm, the *binary search* for some sequences which are ordered in ascending order and for others which are ordered in descending order, there are two possibilities: Either one writes the search procedure twice, one for ascending and one for descending sequences, or one writes a search procedure with a "variable" order relation, which becomes a parameter of the procedure. In this particularly simple case, it is probably best to take the first approach because it avoids the time-consuming transmittal of a parameter procedure at a relatively low penalty in storage space (for storing two search procedures). In large procedures it might be worth while to introduce another parameter in order to avoid duplication of instructions. We will show the use of a Boolean function in the case of the *binary search* procedure:

```
procedure binary search with variable order relation (a, b, l, p, R) ;
        value l, b; integer l, p; real b; real array a ;
        boolean procedure R ;
    begin integer p1 ;
            p := 1 ;
test for end:  if l − p = 1 then go to M ;
            p1 := (l + p) ÷ 2 ;
            if R(a[p1], b) then p := p1 else l := p1 ;
            go to test for end ;
    M :
    end ;
```

The following procedures are examples for Boolean procedures which represent order relations and thus are permissible parameters in the 5th position of the procedure *binary search with variable order relation*:

```
boolean procedure geq (a, b) ;
        geq := a ≧ b ;
boolean procedure leq (a, b) ;
        leq := a ≦ b ;
boolean procedure abscomp (a, b) ;
        abscomp := abs (a) ≧ abs (b) ;
boolean procedure indirectcomp (a, b) ;
        indirectcomp := D [a, 1] ≧ D [b, 1]
```

The last procedure must be defined inside a block where array D is valid; D is a global quantity for this procedure. The comparison of two numbers a and b is here based on the comparison of the first elements in row a and row b of a matrix D.

A Procedure for Merging Two Sequences of Numbers Arranged in Ascending Order

The procedure assumes that the numbers $a[f1]$, $a[f1 + 1]$, . . . , $a[f2]$ are arranged in ascending order, and that the numbers $a[g1]$, $a[g1 + 1]$, . . . , $a[g2]$ are arranged in ascending order. The procedure merges these two sequences into locations $a[h1]$, $a[h1 + 1]$, . . . :

```
procedure merge (a, f1, f2, g1, g2, h1) ; value f1, f2, g1, g2, h1 ;
          array a; integer f1, f2, g1, g2, h1 ;
begin integer f, g, h ;
          comment f, g, and h are "pointers" in the three sequences
               of elements in the array a ;
          f := f1 ; g := g1 ; h := h1 ;
    Q  :  if f > f2 ∧ g > g2 then go to P ;
          if f > f2 then
    M  :            begin a[h] := a[g] ; h := h + 1 ; g := g + 1 ; go to Q
                    end ;
          if g > g2 then
    N  :            begin a[h] := a[f] ; f := f + 1 ; h := h + 1 ; go to Q
                    end ;
          if a[f] ≦ a[g] then go to N else go to M ;
    P  :
end ;
```

Efficiency of the Merge Procedure

Strictly speaking, one cannot judge the efficiency of an ALGOL program unless there is a translator. An ALGOL program which appears to be poor may turn out to be almost optimum if translated by a good translator; and a good ALGOL program may turn out to be very bad if translated by a poor translator. This means, of course, that the "machine independence" of ALGOL has its limitations as soon as efficiency of the intricate kind discussed below becomes important. One may, however, judge the quality of an ALGOL program under the assumption that it is translated by a "simple-minded" translator, that is, a translator which follows the instructions of an ALGOL program very closely without looking for possible savings in instructions. Such a simple-minded translator would produce a program for the procedure *merge* which contains some inefficiencies. The program will make, for example, the comparisons $f > f2$ and $g > g2$ twice, and for such a translator the following program would be better:

```
                    . . .
      f := f1 ; g := g1 ; h := h1 ;
  Q : if f > f2 then begin if g > g2 then go to P else go to M
                 end ;
      if g > g2 then
  N :            begin a[h] := a[f] ; f := f + 1 ; go to R end
                 else if a[f] ≤ a[g] then go to N else go to M ;
  M : a[h] := a[g] ; g := g + 1 ;
  R : h := h + 1 ; go to Q ;
  P :
end
```

This program will still lead to inefficiencies, because for each comparison between elements $a[f]$ and $a[g]$ which is made, it requires:

1. computation of the addresses of $a[f]$ and $a[g]$,
2. getting these elements from memory to the arithmetic unit,
3. computation again, after the comparison, of the addresses of $a[f]$ or $a[g]$ for use in statements N or M, and finally transmittal of the smaller element to its proper place $a(h)$.

The following program avoids some of the inefficiencies:

```
                              . . .
      f := f1 ; g := g1 ; h := h1 ; G := a[g] ; F := a[f] ;
  Q : if f > f2 then
          for i := 0 step 1 until g2 - g do a[h + i] := a[g + i]
  else if g > g2 then
          for i := 0 step 1 until f2 - f do a[h + i] := a[f + i]
  else begin if G ≦ f then
                  begin a[h] := G ; g := g + 1 ; G := a[g] ; h := h + 1 end
            else begin a[h] := F ; f := f + 1 ; F := a[f] ; h := h + 1 end
            go to Q
      end;
```

The last program is about as good as one can get in "optimizing" an ALGOL formulation for this computation process. Such things as register assignment in the arithmetic unit, of course, lie beyond the scope of ALGOL, and for some time to come the optimum use of special registers will be beyond the scope of translators.

A Procedure for Sorting a Set of Numbers

The elements to be sorted are stored in $a[f]$, $a[f + 1]$, ... , $a[f + n - 1]$. The sorting is done according to the now classical procedure by von Neumann and Goldstine (6). Sequences of ordered numbers of length 1, 2, 4, ... are merged to create ordered sequences of twice this size. After, at most, $[\log_2 n]$ sweeps, the original set is sorted. The procedure uses n auxiliary storage locations, namely $a[aux]$, $a[aux + 1]$, ... , where *aux* is one of the parameters of the procedure. At the end of the program the ordered sequence is stored in either $a[f]$ and the following locations, or in $a[aux]$ and the following locations, depending on whether $[\log_2 n]$ is even or odd. The Boolean variable E contains the value **true** in the first case and the value **false** in the second case. The procedure *sort by merge* uses procedure *Min*(*a, b*), which is assumed to be declared somewhere else:

```
procedure sort by merge (a, f, aux, n, E) ;
          array a ; integer f , aux , n ; boolean E ;
```

```
begin integer a1 , a2 , l , j , b ;
          comment
          a1 = f and a2 = aux if the merge works from the original loca-
              tions to the auxiliary locations
          a1 = aux and a2 = f if the merge works from the auxiliary loca-
              tions to the original locations
          l is the length of the ordered sequences
          j counts through the set of ordered sequences;
          comment
          the actual program starts here.  Note how short it is;
          l := 1 ;  a1 := f;  a2 := aux;
      Q : for j := 0 step 2 × l until n − 1 do
              merge (a, a1 + j, a1 + Min (n − 1, j + l − 1), a1 + j + l,
                  a1 + Min (n − 1, j + 2 × l − 1), f + a2) ;
           l := 2 × l ;
           if l < n then begin b := a1 ; a1 := a2 ; a2 := b go to Q end ;
       P : if a1 ≠ f then E := false else E := true
  end ;
```

Use of Special Parameter Delimiters

The program *sort by merge* given above is straightforward and can be easily understood if one knows the meaning of procedure *merge*, which is called by the procedure *sort by merge*. The trouble in understanding the call of procedure *merge* lies in its many parameters, the meaning of which has to be retrieved from the description of procedure *merge*. Although the reader of a program cannot be relieved of the task of looking into the description of the procedures which are used in this program, he may be given some help by the use of "special parameter delimiters." A special parameter delimiter is a string beginning with a closing parenthesis, followed by a string of letters, a colon, and an opening parenthesis. Examples of a parameter delimiter are:

) this is a parameter delimiter : (

) the next parameter must be a positive integer : (

All such special parameter delimiters are equivalent, and they are equivalent to a comma. They may be used to separate parameters in procedure declarations and/or in procedure calls. If a special parameter delimiter is used in the declaration of a procedure, the call may yet use a comma, or even a special parameter delimiter with a different letter string.

With these special parameter delimiters, the call of the *merge* procedure can be made in the following more readable way:

```
merge (a)
      merge the elements from position : (a1 + j1)
                           to position : (a1 + Min(n - 1, j + l - 1))
        and the elements from position : (a1 + j + l)
                           to position : (a1 + Min (n - 1, j + 2 × l - 1))
           into positions upwards from : (a2 + j) ;
comment  the second set of numbers is void in case n - 1 < j + l ;
```

A Sorting Procedure Based on Uniform Distribution of the Numbers to be Sorted

The following procedure is useful only if the numbers to be sorted are almost uniformly distributed between the numbers l (= lower limit) and u (= upper limit). The interval from l to u is divided into $I + 1$ intervals of equal length. In a first "sweep" over the elements $a[i]$ ($i = 1, 2, \ldots, n$), one determines the numbers $C[j]$, $C[j]$ = number of elements in jth interval ($j = 0, 1, \ldots, I$). In a second sweep, the elements $a[i]$ are transmitted into an auxiliary storage region in such a way that for all j and $j1$, if $j > j1$, the elements belonging to interval j are stored after those belonging to interval $j1$. If $j = j1$, the elements are arranged in the original order. In a third sweep, the elements in the different intervals are sorted.

Since the amount of work to be done per element increases in the sort by merge process as $\log_2 n$, the saving which results from this procedure may be substantial if n is large:

```
procedure sorting by distribution counting (a, n, I, u, l, aux) ;
          value n, I, u, l; integer n, I; real u, l, real array a ;
comment   the elements to be sorted are a[1], a[2], ... a[n] .
          I + 1 is the number of intervals
          l and u are lower and upper limits for the a[i] respectively ;
begin     integer array C[-2, I] ;
          comment C[j] will be first used to store the number of
                   elements in the jth interval ;
          for j := 0 step 1 until I do C[j] := 0 ;
          for i := 1 step 1 until n do
```

```
        begin $j$ := entier ($I \times (a[i] - l)/(u - l)$) ;
                $C[j] := C[j] + 1$
        end ;
    $C[-1] := 0$ ;
    for $j$ := 1 step 1 until $I$ do $C[j] := C[j] + C[j-1]$ ;
    comment at this place $C[j-1]$ contains the number of
            elements in intervals below $j$ from here on
            $C[j-1]$ will be used as a pointer for the
            position of the elements in interval $j$ ;
    for $i$ := 1 step 1 until $n$ do
        begin $j$ := entier ($I \times (a[i] - l)/(u - l)$) $- 1$ ;
                $u[aux + C[j]] := u[i]$ ;
                $C[j] := C[j] + 1$
        end ;
    comment at this point $C[j-1]$ contains the position
            relative to $a[aux]$ which is occupied by the
            last element in interval $j$ ;
    $C[-2] := -1$ ;
    for $j := -1$ step 1 until $I - 1$ do
        sort the elements from position  :  $(aux + C[j-1] + 1)$
                                to position  :  $(aux + C[j] - 1)$
        so that the sorted sequence appears
                in positions upwards from  :  $(C[j-1] + 1)$ ;
end of procedure sorting by distribution counting
    comment this is a call of a procedure which is not de-
            clared in this paper, but which could easily
            be constructed from the procedure sort by
            merge ;
```

The sorting procedures given here can be generalized, by the addition of a Boolean function as a parameter, to be valid for any order relation.

CHECKLIST OF IMPORTANT ALGOL CONCEPTS

ACKNOWLEDGMENT

Tom Sobasky prepared notes from my talks about ALGOL 60 at the 1960 summer session on "Advances in Programming and Artificial Intelligence," Chapel Hill, North Carolina. An earlier version of this report was published in the proceedings of this summer session. A. A. Grau and L. L. Bumgarner helped in preparing this first version by making valuable comments and corrections. The present version incorporates further suggestions and corrections by W. Börsch-Supan, A. S. Householder, F. L. Bauer, K. Samelson, and W. R. Busing. I wish to thank all of these gentlemen.

LITERATURE CITED

(*1*) A. J. Perlis and K. Samelson, "Preliminary Report – International Algebraic Language," *Communications of the ACM* **1**(12), 8–22 (1959).

(*2*) P. Naur (ed.), "Report on the Algorithmic Language ALGOL 60," *Communications of the ACM* **3**, 299–314 (1960).

(*3*) *Math. Panel Ann. Progr. Rept. Dec. 31, 1960*, ORNL-3082, pp 6–20.

(*4*) J. Jensen and P. Naur, "An Implementation of ALGOL 60 Procedures," *Nordisk Tidskrift for Information – Behandling* **1**(1), 38–47 (1961).

(*5*) A. A. Grau, *The Structure of an ALGOL Translator*, ORNL-3054 (Jan. 23, 1961).

(*6*) H. H. Goldstine and J. von Neumann, *Planning and Coding for an Electronic Computing Instrument*, Institute for Advanced Study, Princeton, N. J., 1947/48.

ORNL-3148
UC-32 – Mathematics and Computers
TID-4500 (16th ed.)

INTERNAL DISTRIBUTION

1.	Biology Library	223.	A. A. Grau
2.	Reactor Division Library	224.	M. T. Harkrider
3-4.	Central Research Library	225.	A. Hollaender
5.	ORNL – Y-12 Technical Library, Document Reference Section	226-250.	A. S. Householder
6-200.	Laboratory Records Department	251.	R. G. Jordan (Y-12)
201.	Laboratory Records, ORNL R. C.	252.	W. H. Jordan
202.	N. B. Alexander	253.	M. T. Kelley
203.	D. E. Arnurius	254.	J. A. Lane
204.	G. J. Atta	255.	J. G. LaTorre
205.	S. E. Atta	256.	M. P. Lietzke
206.	S. R. Bernard	257.	T. A. Lincoln
207.	N. A. Betz	258.	S. C. Lind
208.	F. T. Binford	259.	R. S. Livingston
209.	H. M. Bottenbruch	260.	E. C. Long
210.	L. L. Bumgarner	261.	M. J. Mader
211.	H. P. Carter	262.	K. Z. Morgan
212.	C. E. Center	263.	J. P. Murray (K-25)
213.	E. L. Cooper	264.	M. L. Nelson
214.	A. H. Culkowski	265.	J. J. Rayburn
215.	F. L. Culler	266.	H. E. Seagren
216.	N. M. Dismuke	267.	E. D. Shipley
217.	A. C. Downing	268.	M. J. Skinner
218.	M. B. Emmett	269.	A. H. Snell
219.	M. Feliciano	270.	J. A. Swartout
220.	B. A. Flores	271.	E. H. Taylor
221.	J. H. Frye, Jr.	272.	D. J. Wehe
222.	W. Gautschi	273.	A. M. Weinberg

EXTERNAL DISTRIBUTION

274. Division of Research and Development, AEC, ORO
275-813. Given distribution as shown in TID-4500 (16th ed.) under Mathematics and Computers category