A PROPOSED EXTENSION TO ALGOL 60

by R. W. Hockney

1.0. Introduction

1.1. Whilst Algol 60 is a language in which all processes of numerical analysis can be described, it must be admitted that the description of long algorithms involving complex numbers, matrix operations and the logical manipulation of digit patterns, even with the aid of 'procedures' is strained and cumbersome with a printed appearance unlike that of ordinary mathematics with which the engineer, mathematician or physicist is familiar.

1.2. This fact is likely to limit severely the usefulness of Algol 60 as a common programming language for scientific use in Industry, and the industrial purchaser of computers will undoubtedly demand of the computer manufacturers and their 'translator' writers more convenient ways of performing these operations, before the next review of the Algol language, which is anticipated to be in 1963.

1.3. Unless some attempt is made to standardise on a convenient extension of Algol 60 at the earliest possible moment a 'babel' of different extensions of Algol 60 is likely to arise in the next 2 years which will make the task of providing a universal Algol translator for the next generation of computers much more difficult than it need be.

1.4. This paper presents a natural extension of Algol 60 which it is hoped might form the basis of an agreed extension to the language. It is emphasised that what is proposed in no way changes any of the existing language but only adds to it, so that all algorithms written in Algol 60 are included in the proposed extension and could be translated by a translator written to accept the extended language.

2.0. Methods within Algol 60

2.1. The basic barrier to the natural expression of complex, matrix and 'bit' manipulation in Algol 60 is the restriction of value to be real, integer or boolean. This means that a matrix A or complex number q which might be represented by arrays A[1:n, 1:m] and q[1:2] cannot appear as primaries in an expression and can only be referred to as entities in the parameter lists of procedures.

2.2. The consequences of this limitation are apparent in the three methods of evaluating a simple complex expression within Algol 60 given below. The same techniques could be used for matrix and bit manipulation.

2.3. **Example 1** Evaluate \( P = q(a + bd)(d + e) \) in complex arithmetic

**Method 1** Effectively 3 address coding

```
begin real array P, a, b, d, q, e, f, g[1:2];

procedure complex add (x, y, z); comment complex \( x + y \)
   to z; begin ...... end;

procedure complex mult (x, y, z); comment complex \( xy \) to
   z; begin .......... end;

Programme: complex mult (b, d, f);
   complex add (a, f, f);
   complex add (d, e, g);
   complex mult (f, g, f);
   complex mult (f, q, P)
end
```

2.4. **Method 2** R. Naur has suggested another method where the complex variables and operations could be included as identifiers in the parameter list to a general complex arithmetic procedure. The programme might appear:

```
Programme:

Complex arithmetic (P, equals, q, times, 1 bracket, a, plus, b, times, d, r bracket, 1 bracket, d, plus, e, r bracket) end
```

2.5. **Method 3** Another alternative proposed by Duncan for matrix work could be used for complex arithmetic. The formula to be evaluated is written in string quotes as a parameter to a general complex arithmetic procedure. This amounts to defining within string quotes a new algorithmic language for complex work.

```
Programme: Complex arithmetic ('P = q \times (a + b \times d) \times (d + e)') end
```

It seems to the author, however, more desirable to make an explicit extension to the language as proposed in this paper.

2.6. **The Proposed Extension**

3.1. The extension rests on the introduction of two new value types, namely complex and array. A simple variable, that is to say an unscripted identifier, may now be declared to represent a complex number pair, or a whole array (in particular a matrix) of any dimension and size. By the existing rules for arithmetic expressions (para. 3.3 Algol 60) the primaries may now represent complex numbers or whole matrices and be connected by the usual operator symbols of mathematics, i.e. +, -, \( \times \), etc.

3.2. The new concept of array type is very general, in that the elements of an array may be of any type, in particular, they may be themselves of type array. All the basic constituents of an array must however be of the same primary type (i.e. real, complex, Boolean, or integer). For example, the elements of a Boolean array may be a mixture of single Boolean numbers, Boolean arrays of various dimensions and even Boolean arrays of arrays of arrays of various dimensions. It may not however contain any real, complex or integer type elements.
3.3. A simple example of an **array**, whose elements are themselves **arrays**, is a partitioned matrix. An unsubscripted identifier may represent a partitioned matrix and appear in arithmetic statements connected by the usual operations of +, -, ×, etc.

3.4. Another example of **arrays** of **array** occurs in data processing where a one dimensional **array** of **Boolean** digits (that is to say a digit pattern) represents some piece of information. Such a **Boolean** **array** may be represented by an unsubscripted identifier. It may be desired to include an unsubscripted variable representing a list of such information in an expression. This may be done by defining another **Boolean** **array** whose elements are now themselves **Boolean** **arrays** which may be of varying dimensions if the individual items of information require different numbers of digits for their representation. In a similar manner a collection of such lists all of varying lengths can be included in an expression as an unsubscripted identifier by defining another **Boolean** **array**. The process may continue indefinitely, restricted only by limitations imposed by the implementor due to the finite size of computing machines.

3.5. In order to have access to every element of an **array** it is necessary also to generalise the subscript notation. An identifier may now be followed by as many subscript lists as are meaningful. The subscript lists are read from left to right. The first encountered specifies the element of the **array** defined by the identifier. If this element happens to be an **array**, the second subscript list specifies the element of this array. Again this element may be an **array** and a third subscript list specifies the element of this **array**, and so on. When the last subscript list has been used the element found may be of **primary type** or **array type** depending on the number of subscript lists given. Subscripted variables of either type may appear in expressions in the usual way.

3.6. The distinction in Algol 60 between type declarations (para. 5.1) and array declarations (para. 5.2) is now unnecessary but, in order to keep within the framework of the Algol 60 report, this distinction will be maintained in this paper. The only difference between the declarations is that a type declaration does not state the dimensions of the array whereas the array declaration does.

3.7. This concept of **array type** requires that an **array** carries with it a statement of its dimensions and the dimensions of any of its elements that may themselves be **arrays**. This is not difficult and has been used before in matrix interpretive schemes (e.g. G.I.P. on the English Electric DEUCE). It is particularly convenient, relieving the programmer from the unnecessary task of calculating the dimensions of arrays resulting from operations such as matrix multiplication. One should think of an **array** as the totality of the elements of the array together with all the bound pair lists describing the dimensions of the **array**.

3.8. The question arises as to how the **bound pair list** is attached to an **array**. This may be done in two ways. If the **array** is data all bound pair lists are read in, together with the elements of the array, while obeying some 'read array' procedure, and the array would be introduced by a 'type' declaration. If, however, the elements of the **array** are being calculated in the programme, a bound pair list is inserted every time a new array is defined by making an 'array' declaration. Such a declaration is the same as in Algol 60 (para. 5.2) and associates a name (identifier) with the **array** and states the dimensions of the **array**, which may have been calculated in a previous block if desired.
3.9. Some procedures (notably on matrices) will require that
calculations be performed on the dimensions of an array. A function
designator $\text{dim}(E,n)$ is provided for this purpose and has the value
of the $n^{\text{th}}$ bound in the bound pair list of the Expression $E$.
$E$ in particular, may be a subscripted array identifier, when $\text{dim}(E,n)$
provides the dimensions of the element of the array specified by the
subscripts.

8.10. Concerning the meaning to be attached to operations linking array
variables with other array variables and variables of other types
the attitude has been adopted that, when the choice of meaning is not obvious,
it is better to define a useful meaning to the operation than to leave it
undefined.

3.11. Experience on DEUCE has shown the advisability of providing two kinds
of operation between arrays of numbers. On the one hand there are the
conventional operations of matrix arithmetic such as multiplication and
inversion given in the DEUCE G.I.P. Matrix Programming scheme, and
on the other hand, there are 'element by element' operations or identical
operations on lists of numbers as given in the Tabular Interpretive Scheme.
The proposal for this extension is to define all the existing operations
and functions to apply to arrays and to mean that the operation is
performed on corresponding elements of the same arrays in the way defined
between <primary types>. This definition covers matrix addition
and subtraction but a separate symbol, $\ast$, is required for matrix
multiplication and this is introduced.

3.12. Consider now the possible uses of the type Boolean array. This now
permits the use of a single identifier to represent a digit pattern.
The logical operations of $\land, \lor, \neg, \Rightarrow$, when relating primaries of type Boolean
array will be interpreted to yield as result another Boolean array whose
elements are derived by performing the logical operation on the corresponding
elements of the primaries. This extension of the Boolean expressions
provides a facility in Algol equivalent to, and in fact more general than,
the logical operations built into the basic machine code of all computers
but only available in Algol 60 by the continued use of for loops.
Such a facility is required for the convenient description of digit
pattern recognition and manipulation in translators and data processing
and also for the programming of problems in propositional logic and
switching theory.

3.13. All meaningful matrix operations, such as matrix multiplication,
and transposition, are defined between Boolean arrays with the convention
that logical 'and', $\land$, replaces arithmetic multiplication and the
logical 'inclusive or' $\lor$ replaces arithmetic addition. This facility
is particularly useful for describing operations in Boolean algebra.

4.0. Syntax and Semantics of the Extension

The remainder of the paper will attempt to set out as additions to the
Algol 60 report an exact description of the proposed extension. The
statements containing additions are listed in full below and paragraph
numbers in brackets refer to the Algol 60 report.
(2.3.) DELIMITERS

<declarator>::= own Boolean integer real array switch procedure complex

<arithmetic operator>::= + - | x | + | / | * | ↑

(2.5.) NUMBERS AND ARRAYS

<complex number>::= (<number>:<number>)

<number list>::= <number list>,<number>|<number>|<number list>,
<number array>|<number array>

<complex number list>::= <complex number list>,<complex number>|<complex number>
<complex number list>,<complex array>
<complex array>

<number array>::= ([<bound pair list>] <number list>)
<complex array>::= ([<bound pair list>] <complex number list>)
<boolean list>::= <boolean list>,<logical value>|<logical value>
<boolean list>,<boolean array>|<boolean array>
<boolean array>::= ([<bound pair list>] <boolean list>)

(2.5.2) Examples ([1:5] true, true, false, false, true)
([1:3] true, true, false), ([1:2] true, false), false)

(- 177 : + 07.431 08)

([1:2, 1:3, 1:4] 111, 112, 113, 114,
121, 122, 123, 124,
131, 132, 133, 134,
211, 212, 213, 214,
221, 222, 223, 224,
231, 232, 233, 234)

([1:3] ([1:3] 4, 2.7, 5.9), 10.369, ([1:3] 2, 3, 0.7))

(2.5.3) Semantics

In a complex number the first number of the pair is the real part and
the second number is the imaginary part.

In a number (complex or Boolean) array the number (complex number or
Boolean) list is a list containing all the elements of the array which may be
single numbers (complex numbers or logical values) or number (complex or Boolean)
arrays. The elements of an array are separated by commas. The whole array
is embraced by brackets and between the first left bracket and the first
element is the most significant <bound pair list> giving the bounds of the
subscripts used to specify elements in the array according to 5.2.3.1
Subscript bounds. The elements specified by subscripts associated with the most
significant <bound pair list> are called the most significant elements of
the array.
(2.5.4.) Types

Integers are of type integer, numbers are of type real, complex numbers are of type complex, logical values are of type Boolean. Arrays of the above <primary types> are of type integer array, real array, complex array, Boolean array respectively.

(2.8.) VALUES AND TYPES

The various types (integer, real, Boolean, complex, array, integer array, real array, Boolean array, complex array) basically denote properties of values.

(3.1.) VARIABLES

(3.1.1) Syntax

<subscripted variable>::= <array identifier>[<subscript list>]

(3.1.2) Examples

PRIMULA [4, 6, n-1][2, 6][9, n^2, 3]

(3.1.3) Semantics

A variable is a designation given to a single variable or to an array (c.f. 5.1 ARRAY TYPE DECLARATIONS).

(3.1.4) A subscripted variable is an array identifier followed by as many subscript lists as are meaningful, each subscript being enclosed by subscript brackets. The subscript lists are read from left to right and are best defined by reference to the numerical value of an array (para. 2.5 and examples 2.5.2). The first to be encountered is the most significant subscript list and is associated with the bound pair list immediately following the first left hand bracket of the array. The values of the subscripts specify an element of this array according to the following rules. The elements of an array appear as a list separated by commas. When a left bracket appears in the list everything between this bracket and its matching right hand bracket is considered a single element. Such an element will be an array or complex number.

The n^{th} element in the list has subscripts related by the formula

\[ n = i_m - l_m + 1 + \sum_{k=1}^{k=m-1} (i_k - l_k) \prod_{r=k+1}^{r=m} (u_r - l_r + 1) \]

where \( i_r \) is the value of the r^{th} subscript counting from left to right
\( l_r \) is the lower bound of the r^{th} subscript counting from left to right
\( u_r \) is the upper bound of the r^{th} subscript counting from left to right
and the m^{th} subscript is the last.

For the case of a two dimensional array (or matrix) this ordering corresponds to the usual listing of elements row by row.
In the case that the element selected is an array the next subscript list read will specify similarly an element of this array.
The process of selection proceeds in this manner until all the subscript lists have been used. At this stage the selected element may be of
<primary type> or <array type> depending on the number of subscript lists present.

(3.1.4.4.) Example

If A is ([4:5, 1:2] 4, 9.6, ([1:2, 6:8] 12, 3, 5, 10, ([1:4] 0.5, 6, 7), 8), [1:2]4, 7, 1.8), 0.7)
then A[5,1] [2,7] [3] has the value ([1:2]4, 7)

(3.2.) FUNCTION DESIGNATORS

(3.2.3.) Semantics

Function designators define single numerical or logical values,
or values of type <array>, which result through the application of given
sets of rules defined by a procedure declaration (c.f. Section 5.4
PROCEDURE DECLARATIONS) to fixed sets of actual parameters.

(3.2.4.) Standard functions

The expression E in the argument of a standard function may also
be of type complex and the result will have value of type complex with the
usual definition of complex algebra except in the case of sign(E) which
operates on the real part of E only.

The expression E may also be of type integer array, real array or complex
array when the result will be an array each element of which is obtained
from the element of the array E by the rules previously given for operation
on types integer real and complex.

The following functions operate on an expression of type complex or
complex array only

rlpt (E) - type real, the real part of the complex number E
impt (E) - type real, the imaginary part of the complex number E
arg (E) - type real, argument in radians of complex number E
0 ≤ arg (E) < 2π
conj (E) - type complex, complex conjugate of complex number E

The following function operates on expressions of any type of array.

dim (E,n) - type integer, the n\textsuperscript{th} bound (lower and upper counted
separately) of the most significant bound pair list of E.
The following functions operate on 2 dimensional (i.e. Matrices) expressions of type real array, integer array or complex array only.

**Inv (E)** - type same as E or real array if E is an integer array, inverse of the matrix E, square arrays only.

**Trans (E)** - same type as E, transpose of matrix E.

**det (E)** - same type as the elements of E, determinant of E. Square arrays.

**I (E)** - same type as E, the unit matrix with the same dimensions as the most significant bound pair list of E. Defined for square arrays only.

**true (E)** - this function operates only on a Boolean array and results in the logical constant true, if every element of the array is true otherwise the result is the logical constant false. Its most frequent use will be in if clauses where Boolean arrays are determining the course of action.

The functions Trans (E), det (E) and I(E) are also defined for expressions of type Boolean array. The result of the operation is defined by substituting the logical 'and', A, operation for multiplication and logical 'or', V, operation for addition, and the logical constant 'true' for unity and the logical constant false for zero.

### (3.3.) ARITHMETIC EXPRESSIONS

#### (3.3.1.) Syntax

`<multiplying operator> ::= × | / | + | *`

`<primary> ::= <unsigned number>l<variable>l<function designator>l
  (arithmetic expression)l<complex number>l<number array>l
  <complex array>`

#### (3.3.2.) Examples

**Primaries**

-177: + 07.43108

[(1:2, 1:2) (1:3), (4:5:9), (6:3: - 4.9), (+ .02:124)]

#### (3.3.3.) Semantics

An arithmetic expression is a rule for computing a numerical value or an array of values.

#### (3.3.4.) Operators and Types

Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types, real or integer or complex or array.

#### (3.3.4.1.)

When the operands are of type real, integer or complex the operators +, -, × have their conventional meaning (addition, subtraction, and multiplication). The type of the expression will be integer if both the operands are integer otherwise real unless one or both operands are complex type when the expression is complex type. If both operands are
of type array with the same bounds the result of the operations +, -, x, is another array whose elements are obtained by performing the operation as previously defined to corresponding elements of the operands. The operation * is only defined for operands of type array with 2 dimensions, and also for operands of type real, integer or complex. If both operands are array; and have compatible dimensions in the sense of matrix multiplication the result is a real, integer or complex array obtained by this operation. If one of the operands is real, integer or complex, and the other an array the result of +, -, x, is an array obtained by operating on each element of the array with the real, integer or complex operand. The result of * when one (or both) operands is (are) real, integer or complex is identical with x.

Examples (I stands for unit matrix, λ a scalar, A & B matrices).

**Mathematics**

A - λI

**Algol**

A - λI

λA  

A - B

\[ \text{The operator/ is also defined when one or both operands are complex, when it yields a complex result of the division in the sense of complex arithmetic.} \]

The operations /, + are defined for operands of type array if the arrays have identical bounds and the operation is allowed between single elements of the array. The result is an array each element of which is obtained by performing the operation on corresponding elements of the operands.

The operations are also defined if one operand is an array and the other of type real, integer, complex in the same manner as for the operator x in paragraph 3.3.4.1.

For paragraphs (3.3.5.) and (3.3.5.1.) see page 12.

### BOOLEAN EXPRESSIONS

#### Syntax

- **Boolean primary** := logical value | variable | function designator | relation | (Boolean expression) | Boolean array
- **Boolean factor** := Boolean secondary | Boolean factor x Boolean secondary
- **Boolean term** := Boolean factor | Boolean term V Boolean factor | Boolean term + Boolean factor

#### The operators

In the case that the operands are of type Boolean array, with the same bounds, the result of any operation is another Boolean array obtained by performing the operation on corresponding elements of the operands. The relational operators may operate on arithmetic expressions of type array provided the operands have the same bounds when the result is a Boolean array each element of which is obtained by operating on corresponding elements of the operands. In the case of a complex array the relations are applied to the moduli of the elements of the array. The elements of the array are evaluated before the modulus is taken.

An if else Boolean expression is undefined if the simple Boolean expression is false.
The operations denoted by $\Lambda$ and $\times$ are identical as are the operations denoted by $V$ and $\ast$. The operation denoted by $\ast$ is defined as in paragraph 3.3.4.1 but with logical 'and' replacing multiplication and logical 'or' addition.

(3.4.6.1.) According to the syntax given in 3.4.1, the following rules of precedence hold.

1. arithmetic expressions according to section
2. $<\leq \geq \neq$
3. $\neg$
4. $\Lambda \times \ast$
5. $\lor$
6. $\Rightarrow$
7. $\exists$

(4.1) CONDITIONAL STATEMENTS

(4.5.3.1.) If statement

When the Boolean expression is of type $\text{Boolean array}$ the if statement just defined is executed using the most significant elements of the array in turn until the array is exhausted. If an element is itself an array its value is that obtained by applying the function true $(E)$ to the array.

(4.6.) FOR STATEMENTS

(4.6.4.3.) While element

When the Boolean expression $F$ is of type $\text{Boolean array}$, while $F$ shall mean 'while every most significant element of $F$.' If an element is itself an array its value is that obtained by applying the function true $(E)$ to the array.

(5.) DECLARATIONS

(5.1.) TYPE DECLARATIONS

`<primary type> ::= \text{real} \mid \text{integer} \mid \text{Boolean} \mid \text{complex}
`<array type> ::= <primary type> | array
`<type> ::= <primary type> | <array type>`

(5.1.2.) Example integer array $V$

(5.1.3.) Semantics

An identifier declared to be of array type, may appear in expressions either unscripted or subscripted.

When unscripted it is thought of as a single entity comprised of all the elements that form the array together with the bound pair lists describing the dimensions and size of the array (see 5.2).
Because the 'bound pair list' is part of the array when it is declared by a type declaration no 'bound pair list' statement is made. The 'bound pair list' information will be read with the array of which it is part, while obeying a read array procedure.

If the array identifier is subscripted in a way consistent with its 'bound pair lists' the identifier takes the value of the appropriate element of the array.

The type array is synonymous with the type real array.

(5.2.) ARRAY DECLARATIONS

(5.2.3.) Semantics

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts and the types of the variables. The 'bound pair list' is attached to the array by this declaration.

(5.2.5.) The identity of subscripted variables

The identity of a subscripted variable is not related to the subscript bounds given in the array declaration or contained in a quantity of type array.

The following examples are added to those in the Algol 60 report.

Example 3. Evaluate the expression \( P = q(a + bd)(d + e) \) in complex arithmetic.

\[
\text{begin complex } P, q, a, b, d, e; \\
P := q \times (a + b \times d) \times (d + e) \text{ end}
\]

Example 4. Evaluate the formula occurring in J. H. Argyris' theory of structural analysis - 

\[
F = F_0 - b_0^t f b_1 (b_1^t f b_1)^{-1} b_1^t f b_0
\]

where all letters represent partitioned matrices

\[
\text{begin array } F, F_0, b_0, f, b_1; \\
F := F_0 - \text{Trans}(b_0)^t f b_1 \text{Inv}(\text{Trans}(b_1)^t f b_1) \\
\text{ } \times \text{Trans}(b_1)^t f b_0 \text{ end}
\]

Example 5. Compare a digit pattern 'C' with a list of digit patterns \( b_1, b_2, \ldots, b_9 \). If \( C \) is the same as any of the list do Statement 10.

\[
\text{begin Boolean array } C, b; \text{ comment the array } b \text{ is a list of arrays similar to } C; \\
\text{for } i := 1 \text{ step } 1 \text{ until } 10 \text{ do} \\
\text{if } \text{true}(C = b[i]) \text{ then } S \text{ end}
\]
ACKNOWLEDGEMENTS

The author wishes to express his thanks to Mr. N. Woodger of N.P.L. and Messrs. F. Duncan, K. Metzer, B. Randell and E. A. Richards of the English Electric Co. for their helpful criticism of the manuscript. Particular thanks are due to Mr. Richards for the extreme care he has exercised in scrutinising the manuscript in detail and pointing out most of the errors.

ADDENDUM

(3.3.5) Precedence of operators.

first  ↑
second  × / + *
third  + -

CORRIGENDUM

Page 10 Line 7 should read

first arithmetic expressions according to section 3.3.5.