3. DECLARATIONS

3.1. GENERAL

An ALGOL program may be broken into logical segments called blocks which are complete and independent units. Their structure is discussed in Section 8. One important property of a block is that, at the beginning of the block, all of the local entities that are to be referenced inside the block must be declared. Declarations determine how the compiled program will treat certain of its elements; thus it is necessary to precede the use of an identifier with a declaration of type. An identifier may appear in only one declaration within a block; however a block may contain blocks within itself (as shown in 8.3). Any of these blocks may declare variables taking on names used in outer blocks, thus redefining them for the inner block.

3.2. TYPE DECLARATIONS

The type declaration defines the type of variable named by an identifier. This declaration specifies that all values which the identifier assumes must be of the designated type. The general form of type declaration is:

$<$ type $>$ $<$ type list $>$

where $<$type list$>$ is a list of identifiers separated by commas. Each declaration is terminated by a ; or $. The five possible type declarations are:

INTEGER – Integral values represented internally by 36 bits. The range of an integer (in magnitude) is zero thru $2^{35}-1$ inclusive.

REAL – Floating point numbers internally represented by 9 bits for sign of the number and the exponent and 27 bits for the fraction. The range of a real number (in magnitude) is 10^{-38} to 10^{38} and 0 with approximately 8 digits of precision. Any real quantity which is less than 10^{-38} is represented by zero.

REAL 2 – Floating point numbers internally represented by 12 bits for sign of the number and the exponent and 60 bits for fraction. The range of a REAL 2 number (in magnitude) is approximately 10^{-308} to 10^{308} and zero with approximately 18 digits of precision.

COMPLEX – Complex values of the form A + i*B where A and B are **REAL** numbers.

BOOLEAN – Truth values, **TRUE** or **FALSE**.

Examples of type declarations are:

```
INTEGER    I, J, K, COUNTER $
REAL       X, Y, TEMP, VELOCITY $
BOOLEAN    AJAX $
COMPLEX    Z1, Z2, U, V $
REAL 2     A, B, C $
```

3.3. ARRAY DECLARATIONS

When declaring simple variables as described above, a different name must be used for each different variable being defined. The **ARRAY** declaration provides a means of referring to a collection of numbers that fall into a matrix or array by the use of a single identifier.

This **ARRAY** declaration specifies to the compiler the structure which is to be imposed on this collection. An array is a group or set of elements arranged so that each may be identified by its position within the group. The compiler considers all elements of array to be of the same type.

Arrays in this compiler are restricted to those of rectangular construction in n-dimensional space.

For example, the declaration **REAL ARRAY** A(1:10) defines an array A with ten elements which may be referenced by:

```
A(1) A(2) A(3) A(4) A(5) A(6) A(7) A(8) A(9) A(10).
```

The general form of array declaration is

< type >**ARRAY** < array list > (<bound pair list>)

where type may be of any of the types given in 3.2. If type is omitted, it is assumed to be **REAL**. The array list specifies the names of the arrays. The bound-pair list consists of a bound pair for each array dimension. Each bound pair is of the form:

lower limit:upper limit

A complete array declaration for a single array is of the form:

ARRAY A l_1:u_1, l_2:u_2, l_3:u_3, --- l_n:u_n.

Where l's represent lower bounds and u's represent upper bounds. Either or both of the bounds may be negative, but $l_i \leq u_i$.

For example:

```
INTEGER ARRAY I(0:4, 1:3)
```

defines an array composed of five rows and three columns of integers as follows:

```
I(0,1)     I(0,2)     I(0,3)
I(1,1)     I(1,2)     I(1,3)
I(2,1)     I(2,2)     I(2,3)
I(3,1)     I(3,2)     I(3,3)
I(4,1)     I(4,2)     I(4,3)
```

In the previous declaration, the parts 0:4 and 1:3 are called bound pairs, and each set of them defines a subscript position. The first digit of the bound pair specifies the lowest possible value for that subscript position, and the second specifies the highest. An element of array I is referenced by the identifier I followed by a subscript list enclosed in parentheses (see 4.2.3.1). Since the lower bound of the first subscript position is 0, I(3,2) refers to the element in the fourth row and second column of array I. There is no limit to the number of subscript positions an array may have. However, declarations like

```
REAL ARRAY A(6:5) $
```

are not allowed; since the lower bound must not be greater than the upper bound.

Array identifiers of the same type, separated by commas, may be included in one declaration:

```
BOOLEAN ARRAY A(1..2), B(1:10,14:22), C(-2:7,0:100)
```

If two or more arrays are of the same type and same size, they may be listed sequentially with the dimension specification after the last array identifier in the group.

```
COMPLEX ARRAY COM, COMI,DECOM,COMCONJ(3:10) $
```

This declaration defines four one-dimensional arrays. Each consists of eight complex numbers and the subscripts of the elements range from three to ten.

One of the most important features of ALGOL is that the expressions for the bound pairs need not be constants; they can be any meaningful expressions (see Section 4).

Example:

```
REAL ARRAY A(1:N,I//2:ENTIER(X),0:TIMEMAX),DP1,DP2(-INFINITY:
           DP1,DP2(-INFINITY:INFINITY) $
INFINITY) $
```

The size of these arrays depends on the values of N, I, X, TIMEMAX, and INFINITY. Therefore, the size varies from one execution to another. Because of this, the actual storage cells for the array are allocated during execution each time the block (in which the array declaration occurs) is entered; i. e., at the place the array is declared. This is called 'dynamic' storage allocation. All the variables in a program except own variables (see 3.5) are allocated in storage in this way. Section 8 explains the process of allocating variable storage for ALGOL. Note that the dynamic allocation concept cannot be used in the outermost block (i. e., the bound pair list may contain only constants in the array declarations in the outermost block).

3.4. **STRING** DECLARATIONS

The **STRING** declaration provides a means of referring to a collection of alphanumeric characters in Fieldata code by the use of a single identifier, and at the same time specifies to the compiler the structure which is to be imposed on this collection. The string declaration defines the name and length of the string:

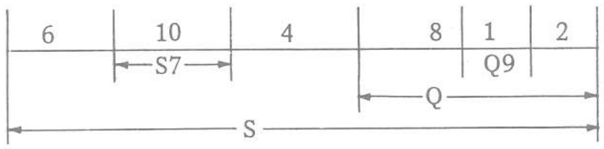
```
STRING S(80) $
```

Strings may have substrings, either named or unnamed:

```
STRING S(L(40),R(40)) $
```

defines a string S as having a length of 80 characters with the first 40 characters being a string L and the second 40 a string R.

```
STRING S(6,S7(10),4,Q(8,Q9(1),2)) $
```



The above declaration defines the strings S, S7, and Q and Q9. It also gives their relative position since 6, 4, and 2 are unnamed substrings. The expression for the length of a string must be positive and less than 4096. Strings, like simple variables and arrays, may be declared with an identifier list:

```
STRING CARD (80), LINE(132), ITEM(CODE(DEPT(2), SECTION (8)),
        5,NAME(30), RATE(5), TIME(5), GROSS(10), NET(10))$
```

The string CARD holds 80 characters corresponding to a card image. Correspondingly, the string LINE holds one print line image. The string ITEM, on the other hand, has the somewhat complicated structure shown below:

DEPT(2)	SECTION(8)			Rate	Time	Gross	Net
CODE(10)		(5)	NAME(30)	(5)	(5)	(10)	(10)
ITEM(75)							

ITEM has 75 characters partitioned into the strings CODE, NAME, RATE, TIME, GROSS, and NET. In addition, the string CODE of 10 characters is partitioned into the strings DEPT and SECTION. Thus

ITEM(8) ≡ CODE(8) ≡ SECTION(6).

3.4.1. String Arrays

A combination of the string and array declarations defines a quantity known as a string array. A string array is an array whose elements are strings. The form of declaration is:

STRING ARRAY S(<string part>:<array part>)

where < string part > specifies the length of each element of S (and also defines any substrings just like a string declaration) and < array part > is the list of bound pairs just as for a simple array (see 3.3).

Example:

```
STRING ARRAY S(L(40),R(40):1:10, 1:10) $
```

defines a two-dimensional array S with ten rows and ten columns. Each element of the array is a string of 80 characters. Furthermore each string consists of substrings L and R each 40 characters long. Referencing of substrings is discussed in 4.4.

3.5. **OWN** DECLARATIONS

Whenever a block is entered, the simple variables and arrays that are declared within that block are given the value zero, and strings are given the value (blank) in each of their character positions. The additional symbol **OWN** in front of any one of these types of declarations changes this initialization in the following way: the first time the block is entered they are given initial values as above. In subsequent entrances to the block they have the same value as they had on the last exit from the block.

Examples:

```
BEGIN INTEGER I $
      REAL FX, FY $
      OWN BOOLEAN ALPHA,BETA $
      OWN REAL ARRAY DEV (1:10, 1:10) $
```

In general all declarations allowed in 3.2, 3.3, and 3.4 of this chapter are also permitted as OWN declarations. The exception to this rule is that the length of a string or the length of any of the subscript positions of an array does not change after the first entrance to the block. Thus, if a block begins by:

```
BEGIN OWN ARRAY A(0:N)
```

and N has the value six (elements are numbered zero through six), the length of A remains seven throughout the program even if N has a different value at the next entrance to the block.

3.6. DEFAULT DECLARATIONS

The **OTHERWISE** declaration allows the programmer to specify that all simple variables (those without subscripts), unnamed in a type declaration are assumed to be of a given type.

```
BEGIN REAL X, FX, FPX $
      INTEGER OTHERWISE
```

means that any other simple variables besides X, FX, FPX that are encountered in this block are to be integers. The **OTHERWISE** declaration may not be used in connection with an array or string. A hazard of this declaration is that it carries the danger that 'new' variables may be created unintentionally and not noticed.

Example:

```
BEGIN INTEGER OTHERWISE $
      BOOMBOOM = 2$
      AEN      = 4$
BOOMBOOM = ((BOOMBOOM+AEN)*BOOMBOOM+AEN)*BOOMBOM
```

The variable BOOMBOM has crept into the calculation when BOOMBOOM was the proper one. Therefore the **OTHERWISE** declaration must be used with care. Another type declaration may follow the **OTHERWISE** declaration.

3.7. THE **COMMENT**

The **COMMENT** allows the programmer to include such things as clarifying remarks and identifying symbols in the printed compilation. A comment may serve any purpose the programmer desires once it is ignored by the compiler.

Two commonly used forms are:

BEGIN COMMENT < any sequence not containing ; or $ >;
; **COMMENT** < any sequence not containing ; or $ >;

Example:

BEGIN COMMENT SAMPLE PROGRAM USING UNIVAC 1108 ALGOL $

Any characters following an **END** and preceding another **END, ELSE,** or a semicolon are also treated as comments.

Examples:

```
END OF INNER LOOP END OF OUTER LOOP $
END THIS TERMINATES THE THEN PART ELSE
END OF HEAT TRANSFER PROGRAM $
```

3.8. **FORMAT, LIST, SWITCH, PROCEDURE, LOCAL**

The other declarations are of a more complicated nature and appear in other parts of the manual. **FORMAT** and **LIST** are concerned only with input/output and are discussed in Section 9. Procedures are discussed in Section 7 and switches in Section 5. The **LOCAL** declaration is added to the language to allow faster (one pass) translation into object code. It is discussed in Section 8.