Many of us are immersed daily in the often time-consuming process of writing and maintaining statistical programs. It is easy to become so immersed that one never has time to look ahead and think strategically rather than tactically. In this conference we have tried to look at strategy.

Conclusion

I hope these papers will provide an assessment by a special group of computer users of their needs, how far these are met by present-day software (programming languages and operating systems), and what future improvements are required. We believe that there is an urgent need for better communication between language designers and their potential customers. (The recent IFIP Working Group 2.3 set up under the chairmanship of Dr Woodger is another response to the same need.) The designers must know what facilities will be useful to their customers, who in turn must know what they can reasonably ask for. In setting down our ideas in the papers for this conference, we hope simultaneously to persuade language designers to be more helpful to statisticians, and to let statisticians know what is happening and could happen in statistical programming.

References

Anderson, A. J. B. and Lowe, Bridget I. (1970). A multivariate analysis computer program. *Appl. Statist.*, **19**, 18–26.

Chambers, J. M. (1969). A computer system for fitting models to data. *Appl. Statist.*, **18**, 249–263.

Godfrey, M. D. (1971). Operating system considerations for statistical computing. *Appl. Statist.*, **20**, 45–56.

Golub, G. H. (1969). Matrix decompositions and statistical calculations. In *Statistical Computation*, pp. 365–397. New York: Academic Press.

Herraman, Carol (1968). Sums of squares and products matrix. *Appl. Statist.*, **17**, 289–292.

Irons, E. T. (1970. Experience with an extensible language. *Comm. Ass. Comp. Mach.*, **13**, 31–40.

Van Reeken, A. J. (1971). Report on the work of the Dutch Working Party on Statistical Computing. *Appl. Statist.*, **20**, 73–79.

Wilkinson, J. H. (1965). *The Algebraic Eigenvalue Problem.* Oxford: Clarendon Press.

Internal Data Structures

By J. C. Gower and I. D. Hill

Rothamsted Experimental Station *M.R.C. Computer Unit*

Summary

In most computer languages, the data are regarded as being held in machine storage either as individual scalar quantities or as members of rectangular arrays. It would make these languages more useful if the user were allowed to specify the shapes of other structures in which he wished to consider the data as being held, such as triangular arrays, trees, etc.

To go with the ability to define structures the user would require the ability to define operators to act upon structures.

Introduction

In these pages, Nelder and Cooper (1971) discuss how data are converted between external representations and internal forms that are usually based on mathematical

scalar quantities represented by the machine hardware in terms of machine words, combinations of bytes or combinations of bits. In this paper we intend at first to follow the path pioneered by the authors of Algol 60, assuming that all questions of input and output can be ignored. We are concerned only with descriptions of and operations on the data present in core.

The simplest internal data-structure is a single item represented by an identifier name. At any moment during the running of the program the identifier name has an associated address used to access the item. More complicated structures consist of more than one item, each of which can be accessed through a single identifier name, with or without qualification, as described below (see Sections 2.4 and 4). Structures of this kind will be assumed to exist in core store only, unless stated otherwise. The interfacing methods allowing such internal structures to be transferred between core and backing files are nowadays handled mainly by the operating system (Godfrey, 1971). The items of a structure need not, and often will not, be contiguous in core and this may complicate accessing and so diminish program efficiency. In the following the term *structure* stands for internal data structure.

1. Types of Structure

Nearly all languages allow simple scalar variables, and arrays of one dimension (also called "vectors"). These are uncomplicated in concept, are quick to access and serve a great many purposes admirably.

1.1. *One-dimensional Arrays*

The one-dimensional array is indeed the "natural" unit of computer structure (except in some experimental work on two-dimensional stores), because the addresses of storage locations are normally serial integers. Other structures can be implemented only by mapping to this one-dimensional form.

In the days of Mercury Autocode, for example, this was all that was available, and any mapping required had to be done by the user. High-level languages, however, should take this burden from the user's shoulder and do the mapping for him as required, allowing him to act as if the actual structure specified were physically present in the computer. This in no way prevents him from specifying a vector when desired, and applying his own mapping to it.

A particular type of vector is the *string*, which may alternatively be considered as a one-dimensional character array. Its most important use is for the output of text.

1.2. *Multi-dimensional Arrays*

Next we have the multi-dimensional array, sometimes called a "matrix", although some people restrict this word to the two-dimensional case only. Existing languages all seem to require arrays to be rectangular, i.e. the bounds of any dimension must be constant over all values of the other dimensions. The triangular matrix is useful in statistics, but must be mapped from two dimensions to one in an appropriate way.

At least two different ways of mapping multi-dimensional arrays are in use. One accesses the array by multiplying each subscript (except the last) by a constant, and adding to find an address; the second uses tables of pointers, each subscript giving an address in a further table until the final subscript gives the final required address. This second approach is quicker, because it avoids multiplication, but it takes extra storage for the tables of pointers.

Although modern computing languages allow variable array bounds, the specified number of dimensions of an array must be constant. Where a variable number of dimensions is required, the user has to do his own mapping on to one dimension.

In Fortran an array can be accessed as one-, two- or three-dimensional by declaring three arrays of one, two and three dimensions respectively and using an *EQUIVALENCE* statement to make them all, in fact, the same array with different names.

More flexible approaches are possible within existing languages (see, for example, Gower, 1968; Hill, 1971).

PL/1, by use of its *DEFINED* attribute, allows the user to specify, within an array declaration, any mapping required. For example:

DECLARE $A(12)$, $B(4,3)$ *DEFINED* $A(4*(2SUB-1)+1SUB)$;

means that an array A with bounds 1 and 12 is required; it may be referred to either as $A(I)$ or as $B(I, J)$. $B(I, J)$ will be treated as meaning $A(4*(J-1)+I)$.

In this case merely a simple mapping of two dimensions to one has been achieved but it is easy to see how the device can be used for mapping any desired shape to one dimension. The user, of course, has to work out the mapping in detail, but having done so and included the mapping in a declaration, subsequent use is direct and simple.

1.3. *Trees*

The hierarchical tree is a structure that often occurs in statistical surveys and experiments. Yet it is found more often in commercial languages than scientific ones. Cobol and PL/1 both have tree structures as an important part of the language but with restrictions on parametrization of size. They can be dealt with in Algol *W* and Algol 68 by regarding them as special cases of records, and in Pop-2 by regarding them as special cases of lists. However, where a tree is known to be hierarchical it may be dealt with more expeditiously than by a generalized list-processing facility.

Two types of hierarchical structure are important in statistical work. First, there is the type that occurs in survey work, say of the form "Farms within Centres within Districts within Regions". Here there may well be numeric and other information associated with every level of the hierarchy. Secondly, in split-plot experiments and hierarchical analysis-of-variance problems the structures have numeric information only for the units at the lowest level of the classification. In principle, both types can be described by the PL-1/Cobol *STRUCTURE* though there would not normally be sufficient core-store to hold all the data of a large survey as a *STRUCTURE*, nor is this usually necessary. Most users would be unable to describe the structure of a large survey in complete detail but might be prepared to assign maximum sizes for every class (possibly wasting an enormous amount of store).

1.4. *Lists*

Pop-2 and specialist list processing languages have lists as basic structures, but these are really very easy to implement in other languages. Two arrays can be declared, one for the values being listed and the other for integers pointing to the next subscript in the list.

1.5. *Functions*

It might seem odd to include functions as a type of data structure. Yet they should be mentioned under this heading because of the blurred distinction between arrays and functions in Pop-2. Both are regarded as methods of retrieving a value by giving one

or more arguments, the only distinction being that for arrays the values are stored, while for functions they are calculated. But even this is not an absolute distinction, since in Pop-2 one can assign a value to a function and retrieve it later, just as with an array.

2. The Need for Structures and Some Consequences

2.1. *Defining New Structures*

Apart from standard structures, users may wish to define new structures from existing components in a similar way to that in which new operations are defined in terms of standard operators.

The ability to define new structures is useful because it allows elements grouped in the user's mind to be similarly grouped, under a single identifier name, in a program. This helps to clarify thought in writing a program, to improve clarity of appearance of the written program and to avoidance of errors, because various operations on structures may be defined once and used repeatedly rather than being specified many times over. The grouping of elements is also useful in reducing the number of parameters when referencing a procedure.

There is little point in being able to define new structures without being able to define operations on them. Thus when we define a table, we need to be able to define such operations as table addition and multiplication. The operation can be defined either by using a procedure or by defining new operators; these two methods are discussed in more detail in Sections 4 and 5.

2.2. *Methods of passing Structures as Subroutine Parameters*

When procedures are used, the structure names must be acceptable as procedure parameters and this raises questions of the meaning and validity of passing structures by reference, value or name.

For a scalar, these three methods of passing parameters are as follows.

(1) *Reference*: this is the method used in most versions of Fortran whereby an address is passed across. Reference to the formal parameter then causes access to the corresponding actual parameter by indirect addressing. Where the actual parameter is an expression, a working variable has to be set up externally to the subroutine as the actual parameter and given the value of the expression. This evaluation is done only once.

(2) *Value*: this is similar to the reference method where the actual parameter is an expression, except that the working variable is internal to the subroutine. The main difference comes when the actual parameter is a variable. Call by value treats such a variable as a particular kind of expression, evaluates it once, assigns its value to a working variable and ignores the actual parameter thereafter. Thus a "value" parameter cannot be used to return a result from the subroutine.

(3) *Name*: this Algol 60 concept interprets a formal parameter exactly as if the actual parameter had been written instead at each place of mention; thus if the formal parameter occurs on the left-hand side of an assignment its corresponding actual parameter cannot be an expression. At every mention of the formal parameter either the address or the value of the actual parameter is re-evaluated, according to context. This can be inefficient, calling for much "housekeeping", but for any program used once only (as is common in scientific research), the convenience for the programmer far outweighs the inefficiency.

In Algol *W* there is a further method of passing parameters, known as "call by result". This is the equivalent, for output from the procedure, of "call by value" for input to the procedure. An internal working variable is used until exit from the procedure, when its value is assigned to the actual parameter.

Algol 68 in principle allows only call by value, but its reference mechanism allows calls by reference or by name to be manufactured, by making the value that is passed be the value of a location, or the value of a procedure.

For structures, in most existing languages, the actual parameter can only be a simple structure identifier; an expression is not allowed. This means that call by reference and by name amount to the same thing, whereas call by value or by result means using extra storage in completely copying the structure, and there is less to be gained from doing so than there is with scalars. Algol *W* indeed does not allow an array to be called by value or by result.

Once operators to act on structures are permitted, structure expressions become usable as parameters and all types of call become possible just as for scalars.

2.3. *Input/Output of Structures*

Just as we wish to use structures as procedure parameters so we may also wish to use them in input and output. It would be unacceptably inflexible to insist that the items comprising a structure must be in a predefined standard order on any external medium. Just as statisticians, using pencil and paper, may copy data received in various formats on to a computing sheet laid out in a standard way for a standard calculation, so must structure input/output procedures provide for describing various patterns of the data items and, acting on this description, transfer each item to its correct internal/external position. This type of facility is, of course, quite separate from the Fortran type of format description which is concerned mainly with numbers of digits, decimal place positions, etc. in scalar items. Nelder (1969) discusses in detail the various forms of output that may be required for structures consisting of two one-way arrays with labelling, two two-way tables with labelling and all m-way marginal tables from an n-way array with labelling.

2.4. *Accessing Structures and Sub-structures*

Structures are defined in terms of basic items or items that are themselves other previously defined structures. Although we have stressed the advantage of being able to refer to a complex structure by a single identifier name we may often require to access only a portion of a complete structure. This means that it must be possible to restrict reference to a structure to include only the desired portion. Most languages that allow this type of restricted reference (Algol 68, Algol *W*, PL/1) do not allow parametric representation, that is as if $X[1, 3]$ were allowed but $X[i, j]$ were not. The reason is that when structures contain items of mixed mode and type (see Section 3) the compiler cannot always determine the type of an item referred to parametrically and so cannot compile efficient object code; it has to allow for all possibilities and incorporate a switch to the correct code for the actual modes and types the parameters take during run time. This would usually be considered unacceptable, but it is equally intolerable for the programmer to have to avoid parametric forms when he knows that all, or a known range of, parameter values must give rise to (say) floating point operands.

Wirth and Hoare (1966) say that a record differs from an array "in that the types of the fields are not required to be identical, so that in general each field of a record may

occupy a different amount of storage. This, of course, makes it unattractive to select an element from a record by means of a computed ordinal number, or index". To say the least, we question the "of course" in that passage.

They also say, "Many applications for which record handling will be found useful are severely limited by the speed and capacity of the computers available. It has therefore been a major aim in the design of the record-handling facilities [in Algol *W*] that in implementation the accessing of records and fields should be accomplished with the utmost efficiency, and that the layout of storage be subjected only to a minimum administrative overhead." That sounds more like the real reason. We wish to emphasize that it is often worth using extra machine time for the sake of human convenience, but there are clearly limits beyond which such a policy cannot reasonably be taken. However, Algol *W* does allow the required parametric representation indirectly as the user can define an array of references.

Often (e.g. Boolean operations) the written form of operator implies that the operand types are known and so the compiler could accept the programmer's judgement as correct; however, there are difficulties when the "extension of operators" principle is available (see Section 5). The main difficulty comes with the standard arithmetic operators $+, -, *, /$, which may be used with either integer or floating point operands (but notice Algol $\div$ which implies integer operands only). Perhaps it would be worth telling the compiler what kind of + etc. was intended, perhaps by using different symbols in ambiguous situations.

User-defined structures are often to be thought of as the combination of other structures each with its own name. These sub-structures may often be operated on in their own right, and with dynamic storage allocation their core store positions may be unknown. It turns out that this is no problem, provided that the system keeps track of the addresses of the components of each sub-structure; this is done by using pointers. Pointers are often private to the internal workings of the system but to allow user-defined structures they must be available at the user-language level. Modern languages allow this (e.g. PL/1, Algol 68). With older languages, the user has to construct, with some inconvenience, his own pointers to regions within a one-dimensional array area reserved for this purpose.

One implication of having pointer-defined structures is that the elements of a structure need not, and usually do not, occupy a contiguous region of core store. This is an essential property when two or more structures refer to some of the same sub-structures. That the core region occupied by a structure is disjoint need not seriously affect the efficiency of accessing the structure elements. This efficiency depends on the access pattern; if it is regular, successive elements can usually be obtained by adding some simple function to the previous address and little is lost, but if irregular, every address has to be recalculated *ab initio* and this is time consuming.

3. Types, Modes, Representations and Structures

These concepts are much confused, and unfortunately the names chosen by different languages for them are inconsistent.

For present purposes we use "type" to indicate the basic content of the data, such as numerical, Boolean, character, etc. Within "type" we use "mode" to indicate the form in which the data are held; thus integer or floating point are modes, within the numerical type. Some types have only one mode. "Structure" indicates the manner in which several data items are related to each other. Fortran can separate the structure information of an array (declared by a *DIMENSION* statement) from its type, which

may be specifically declared or set by default according to the initial letter of its identifier.

Perhaps another word "representation" should also be introduced. Thus integer and real are different modes and, within each, single length and double length could be different representations.

It is regrettable that existing languages use the words "type" and "mode" with some abandon. Algol 60 uses "type" for two of these three concepts—thus integer is regarded as a type rather than as integer mode within numerical type, and quite often even integer array is regarded as a type rather than as a structure of integers, although the Algol 60 Report does not in fact endorse this usage. "Array", which should describe the structure independently of the type, is unfortunately taken to be merely a synonym of "real array".

A switch in Algol 60 declares a one-way array of elements of type label although label is not strictly an Algol type. Furthermore, the values can be assigned to a switch only within a declaration, a facility not available for any other kind of element.

Algol 68 uses the word "mode" instead of "type", but without any real change of meaning to go with the change of word.

By separating the concepts of type, mode and structure, it should become possible to define a structure independently of the types and modes to go into it. Thus a triangular matrix, for example, could be defined merely by its shape and means of access. A real triangular matrix, or a Boolean triangular matrix, for example, could then be declared without further ado.

We have investigated the question of producing a table to show which facilities are available, and which not, in the various computing languages. We have regretfully concluded that such a table is impracticable, for many of the points of interest are concerned less with individual features than with the cross-classification of these features.

Thus within the numerical type, we should wish to tabulate for each language an hierarchical split into real and integer modes, cross-classified with length of representation, cross-classified with the ability to form arrays, to form further structures, to use these as parameters of procedures called using the various passing mechanisms, etc. To get a meaningful table seems to call for something like six dimensions.

Further, within this table it would not suffice merely to indicate "yes" or "no" for each combination of language and features, because so often the only fair answer would be "yes, but there are severe restrictions" or "no, but there is an easy way of getting the same effect".

For example, are character variables available in ASA Fortran? No, but integer variables may be given "Hollerith" values, which comes to much the same thing. Do label arrays exist in Algol 60? Yes, in the form of a "switch", but these are restricted to one dimension, and cannot be assigned to, but only preset.

3.1. *Structures with Mixed Types*

The older languages, Fortran IV and Algol 60, allow an array to contain only one type of variable. Thus, for example, real arrays or integer arrays are allowed, but no array can have some elements of real type and some of integer type although the *EQUIVALENCE* facility in Fortran indirectly allows types to be mixed.

PL/1 allows mixed types within *STRUCTURES* and within arrays by means of the *UNSPEC* facility. Algol *W*, followed by Algol 68, introduces the idea of a record, which in some respects is equivalent to an array in which types may be mixed, although

the mixture must be identical for each record of a given record class and settled at compile time. Thus, if record 1 or a particular class is of the form "integer, integer, real, Boolean, reference, string", so must be all other records of that class.

The records in a class may be thought of as a two-dimensional array, one dimension being of fixed length and a fixed mixture of types as shown, while the other dimension has no fixed bounds or any particular ordering and consists of the individual records.

The accessing of these records is by reference variables, and, because arrays of such references may be formed, complicated patterns are possible. For example the following portion of Algol *W* program

reference (*person*) *A*, *B*, *C*;
record *person* (**integer** *age*; **reference** (*person*) *father*);
age (*A*) := 93; *age* (*B*) := 58; *age* (*C*) := 21;
father (*B*) := *A*; *father* (*C*) := *B*;

will set up a table as follows:

person	**age**	**father**
A	93	
B	58	*A*
C	21	*B*

The expression

age (*father* (*father* (*C*)))

for example will then produce the value 93.

Algol 68 additionally allows mixed arrays whose elements may be any of a nominated set of existing types; these are termed a new type (or mode, in Algol 68 notation) referred to as the union of the constituent types. An array of this new type can be formed, and any types within the union can then be placed at any point in the array. This carries the penalty of run time type checking.

Pop-2 simply evades the whole question by not having any type concept at all. Any sort of data can be stored at any location (which must considerably handicap speedy running).

3.2. *Dynamic Type and Mode Detection*

Flexible handling of structures requires the run time detection of type and structure information. Algol 68 proposes that the user should detect type by comparing the identifier of unknown type with a variable or constant of known type, returning a Boolean true/false result. This is cumbersome when many comparisons have to be made before agreement is found, and it has been suggested that it might be preferable to assign an integer to each type (the user assigning his own suitable integer to any new types he defines) and return the value of this integer. This can be done ideally by defining an integer function of any type of identifier returning the appropriate value. Another method, using the **case** construction, is suggested in an example given by Peck (1970).

A useful structure information function is the dimension operator ρ of APL (Falkoff and Iverson, 1968). If A is an n-way array $\rho(A)$ is a vector of length n whose ith element is the number of levels of the ith dimension. Therefore $\rho\rho(A)$ is a one-element vector whose value is n. If A is a scalar $\rho(A)$ contains no element and $\rho\rho(A)$ is zero. Such operators seem essential for handling user-defined structures (Gower, 1969).

It would also be useful to have a function that determined whether or not the values of a structure had been set or evaluated; if not, some default action could be taken. This function would be useful in detecting parameters unset in procedure calls because they are not relevant in all instances.

4. Examples

As a means of explaining the type of facility that seems to be required we shall adopt the following notation for referring to one-dimensional pointer arrays. The names of identifiers $a, b, c, d, \ldots$ can be thought of as being the elements of a pointer array p (say). Thus $p[1]$ is another name for a, $p[2]$ another name for b, etc. If a happens to be a one-way integer array and b a two-way real array and c a further pointer array we might refer to elements $p[1][i]$, $p[2][j, k]$, $p[3][i][u, v, w]$, etc. Here $c[i]$ is assumed to be the name of a three-way array. Clearly in principle this method allows parametric access but it is part of no computer language and we are using the notation purely illustratively.

Example 1. *Defining a symmetric matrix by its lower triangular elements*

Suppose we wish to define a symmetric matrix as a lower triangular structure (*ltm*) made up of one-way arrays corresponding to the rows, i.e. each successive array has one more element than the previous one, then we could write in an Algol-like language:

```
structure ltm (a,fault);
pointer array a; integer fault;
begin integer i;
i := fault := 0; j := ρ(a)[1];
for i := i+1 while fault = 0 and i ⩽ j do if ρ(a[i]) [1] ≠ i then fault := 1
end ltm;
```

This sequence of instructions defines the lower triangular symmetric matrix and checks that any representation is of the correct form. An *ltm* is now a new user-defined structure with formal parameter a. This defines the shape of an *ltm*-structure but does not reserve any store until a particular instance x (say) is declared as follows:

```
ltm x (u,fault);
```

where u is a pointer array to the appropriate rows of x. The (i, j)th element of x is written $x[i][j]$. Note that $x[i, j]$ is meaningless because x is defined in terms of a one-way pointer array.

Example 2. *Defining a complex number pair*

```
structure complex (a, b);
real a, b; skip;
```

No checks are necessary here; we borrow the **skip** symbol, for a statement that performs no action, from Algol 68.

Delaring x, y, z as the complex pairs $a+ib$, $c+id$, $e+if$ entails the following:

```
real a,b,c,d,e,f;
complex x(a,b), y(c,d), z(e,f);
```

It might be possible to have a simpler method in which the user simply declares **complex** x, y, z; allowing the compiler to note from the structure definition that each requires two real variables.

Defining complex addition $z := x+y$ is then:

```
procedure complexadd (x, y, z);
complex x, y, z;
begin
z[1] := x[1]+y[1];
z[2] := x[2]+y[2]
end;
```

See also Section 5.2.

Example 3. *Defining a tree*

The pointer notation is particularly suitable here. Suppose a, b, c, d, e, f, g are scalar identifiers at the ends of the tree and that these are hierarchically classified as in Fig. 1. Then u is a pointer array with elements a, b, c; x is a pointer array with elements u and d, etc. Thus in an obvious notation we could write

$$u \rightarrow a, b, c;$$
$$v \rightarrow e, f;$$
$$x \rightarrow u, d;$$
$$y \rightarrow v, g;$$
$$t \rightarrow x, y;$$

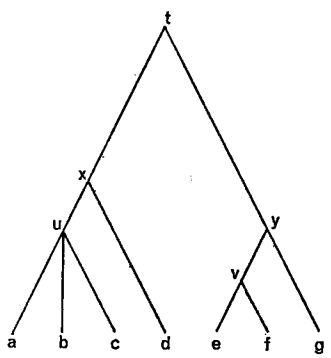

FIG. 1. Example of tree structure.

With our previous notation, a tree is a synonym for a pointer array. Therefore

```
structure tree (a);
pointer array a; skip;
```

is all that is required. Thus with the tree t we can access v by writing $t[2][1]$ and c by writing $t[1][1][3]$. Clearly a, b, c, d, e, f, g could be names of other structures such as arrays. If d is a two-way array for example we might refer to $t[1][2][i, j]$.

Further examples of the notation can be found in Gower (1969). With it any structure shape can be defined once and for all by something like a procedure/subroutine definition with formal parameters. Actual instances of these structures are invoked by associating actual parameters with an identifier name. Values are assigned with separate statements.

5. Operations On Structures

5.1. *Operations using Subroutines*

Provided that structure names can be procedure parameters and that each item of a structure can be accessed, procedures can be written to operate on sets of structures resulting in new structures. This is the method adopted in languages such as Fortran and Algol where array-structures are defined that can be passed as subroutine/procedure parameters but do not have operations defined on them except for their individual scalar components. For this approach to work adequately, access to structures must be in the parametric form; there is little point in defining operations on two-way tables, three-way tables, etc. each with a separate subroutine, with a further $n(n-1)$ subroutines to take care of all combinations of pairs of tables up to order n. It is also unnecessarily tedious if all the internal information on a structure has to be passed as separate specifically declared parameters, as often with Fortran array dimensions.

Access to internal information implies that operations must be available to ask for information on dimensions, array-bounds and element types (see Section 3.2). Consequently all this information must be stored in the compiled program for possible use at run time. With elaborate structures, descriptive information of this type can use as much space (or more?) as the individual items of the structure, so methods for discarding unwanted material must be investigated. Perhaps the compiler can keep a list of those structures for which it compiles interrogative instructions and preserve descriptive material for those structures only.

Thus when adding two lower triangular matrix structures (as defined in Section 4) we would probably first check that both were of the same order and both contained elements of the same type, before embarking on the purely arithmetic part of the procedure; these checks would imply the necessary power to interrogate (see Gower, 1969).

5.2. *Extensible Operators*

An alternative way of operating on structures is to extend the scope of existing basic operators or to invent new basic operators. Thus the symbol + might be extended to operate on arrays or polynomials. This method is neat, when it is available, because it implies that operations on such structures need not be called as procedures, but merely by writing the operator in normal algebraic expressions. The compiler recognizes that the operands are structures for which the operation has been extended and compiles a subroutine call (or the necessary in-line code) to implement the extended operator definition.

Facilities of this kind exist in several modern languages. In Algol 68 the user can define the meaning either of existing symbols or of new symbols, and apply them to existing or to new data types. For example, the operation $a \uparrow b$ is defined only if b is of type integer (unlike the situation in Algol 60). If we wish to use $a \uparrow b$ where a and b are both real, we can define the operation for ourselves as

op $\uparrow$ = (**real** a, b) **real** : **if** $a < 0.0$ **or** $a = 0.0$ **and** $b \leqslant 0.0$
then *trouble* **elsf** $a = 0.0$ **then** 0.0 **else** $exp\ (b \times ln(a))$ **fi**

where "trouble" is the name of a suitable fault procedure.

The "macro" facility of Pop-2 may be used in a similar way. To borrow an example from the *Reference Manual*, a macro may be declared to define the operation associated

with the compound symbol – → as follows:

```
macro – →; vars x y; itemread → x; itemread → y;
macresults ([%" →", y, " →", x%]) end;
```

This leads 7//2 – → q r; to be interpreted as meaning 7//2 → r → q; where // in Pop-2 means integer division with remainder, a function with two results.

If new basic operators are defined their order of precedence must be given. For example in Algol 68 a declaration such as **priority** + = 6 can be used. Incidentally, we think the standard priorities should be given as 10, 20, 30, etc. rather than Algol 68's 1, 2, 3, etc. to allow space for the user to insert his own priorities into the series.

To define the operator + + for complex addition, the procedure *complexadd* defined earlier (Section 4) might be replaced by a declaration of the following kind:

```
op + +(z := x + +y); priority 52;
complex z, x, y;
  begin
  z[1] := x[1]+y[1];
  z[2] := x[2]+y[2]
  end;
```

Similarly forming a complex number from two real numbers using the symbol ,, might be defined as:

```
op ,, (z := x,,y); priority 55;
complex z; real x, y;
  begin
  z[1] := x; z[2] := y
  end;
```

These new symbols could be used in statements like the following:

```
complex a, b, c;
b := 1.0,,0.62; c := 15.584,,1.75;
a := b + +c + +8.1,,6.52;
```

6. Implementation

We have already discussed the effects of some proposals on implementation. Mixed mode/type structures have been shown to lead to inefficient object code if the structure is to be accessed in parametric form, unless special provisions are incorporated.

The feasibility of many of the facilities described has already been demonstrated in languages like APL, BCL, CPL, PL/1 and Pop-2; Algol 68 compilers are being written. What is in more doubt is the overall efficiency of these languages. Computer efficiency cannot profitably be considered without taking other factors into account. For example, in a teaching environment, efficient run time programs are of little value if they take a long time to compile, or are written as an exercise to be run once only. If a program is written to handle a specific problem and the overall time taken to write it can be reduced by thinking in terms of and by handling complex structures, this saving may well be worthwhile when traded against increased use of computer time, even if this involves an interpretive language like APL or Pop-2.

It seems to us that computer efficiency is important only with much-used programs or with programs handling large quantities of data. Standard library programs for routine calculations are a particularly important class and it may well be worth

programming these without reference to complex structures and other, possibly time-consuming, advanced facilities, provided input and output can be made flexible and simple. In other cases the use of advanced facilities may well pay off in terms of human efficiency, the importance of which is often underrated. The whole field of multi-access time-sharing computing although little explored seems a particularly promising one for useful development of internal data structure facilities reflecting the kind of data the user is manipulating.

Missing or unset values are important in statistics. The set/unset attribute of every data-structure and item can be kept by the compiler, but this would entail looking up the appropriate bit every time an item was operated on. To avoid this an extra bit could be associated with every computer word and the necessary checks built into the hardware. This would increase computer costs but might well save money in avoiding nonsense calculations, which often occur when "unset" is recorded by zero or some very large number.

7. Conclusion

Many of the facilities that seem to be required for the convenient handling of statistical structures have already been specified, and sometimes implemented, in various language proposals. That these languages have not been designed with statisticians' needs specially (if at all) in mind, suggests that our requirements are more generally felt than many statisticians realize.

This seems to be the current position. Highly structured data resembling those familiar to statisticians occur in crystallography, structural engineering, business data processing, etc. It is not without interest that in addition to statistics, these are all subjects for which specialized high-level languages have been written. Although these specialized languages have used Fortran (usually) as a source language, Fortran has clearly been felt too cumbersome for general use. This is almost certainly because of its poor data- and structure-handling facilities and because of its inadequate (almost non-existent) extensible features. The resulting high-level languages are excellent for those doing a few stereotyped calculations in very specialized subjects, but are too inflexible for research use or those working slightly outside the area of application for which the language was designed. Anyone wishing to add to such a language must know the system's structure in great detail, so that he can insert new code without clash of identifiers and tie it in with the built-in control and data-handling mechanism. To supply the new code he must also know Fortran, so that he must learn at least two languages.

With the newer extensible languages incorporating data-structure defining facilities, the specialized languages should be unnecessary. The subroutine package approach, which has proved popular on a smaller scale because of its flexibility, could be extended to cope with most, if not all, statistical work. Subroutines could be written for routine statistical calculations and these could be used by unskilled computer users to process standard forms of data, also defined in subroutine form. The parameters in each subroutine would be fewer than is often the case now, because highly structured data would be named with a single identifier. The extensibility feature of standard operators could easily be used, for example, to define all derived variate operations. More skilled users could write new subroutines for their own specialized work or to be added to the standard library used by the less skilled users. They would also intersperse possibly quite elaborate code between calls to standard programs. In this way different centres would tend to build up libraries of programs and data structure definitions suited to their own line of work.

All the users of such a subroutine package would be writing in the same language, so that the unskilled users could build on their elementary knowledge of calling subroutines to become more skilled.

The ease with which standard library subroutines and programs written for a special purpose are combined depends both on the language used and on the computer operating system. A modular language is essential if both identifier clashes, and lengthy and repetitive common-lists, are to be avoided. As the editing and combination of new with previously compiled programs depend on the operating system, efforts should be made to standardize at least on these aspects. It is arguable that these aspects of operating systems should be specified as part of the language design; too often languages are designed as though every program written will be run independently of every other program.

The surest way of getting this sort of language seems to be to band with the non-statistical users and try to influence the language designers to incorporate what is wanted.

References

FALKOFF, A. D. and IVERSON, K. E. (1968). APL/360 *User,s Manual.* IBM Watson Research Center, Yorktown Heights, N.Y.

GODFREY, M. D. (1971). Operating system considerations for statistical computing. *Appl. Statist.*, **20**, 45–56.

GOWER, J. C. (1968). Simulating multidimensional arrays in one dimension. Algorithm AS 1. *Appl. Statist.*, **17**, 180–185.

—— (1969). Autocodes for the statistician. In *Statistical Computing* (J. A. Nelder and R. C. Milton, eds) New York: Academic Press.

HILL, I. D. (1971). Arrays with a variable number of dimensions. Algorithm AS 39. *Appl. Statist.*, **20**, 115–117.

NELDER, J. A. (1969). The description of data structures for statistical computing. In *Statistical Computing* (J. A. Nelder and R. C. Milton, eds), New York: Academic Press. pp. 13–36.

NELDER, J. A. and COOPER, B. E. (1971). Input/output in statistical programming. *Appl. Statist.*, **20**, 56–73.

PECK, J. E. L. (1970). Letter to the editor: Algol 68. *Comp. Bull.*, **14**, 64.

WIRTH, N. and HOARE, C. A. R. (1966). A contribution to the development of Algol. *Comm. Ass. Comp. Mach.*, **9**, 413–432.

Operating System Considerations for Statistical Computing

By MICHAEL D. GODFREY

Department of Mathematics, Imperial College, London

Summary

This paper presents a survey of the main components of computer operating systems, indicates some of the requirements of statistical computing and discusses how some needs of statistical computing might be better met.

A main conclusion of the paper is that, since the processing of user programs through the computing system is itself a statistical computing task, if the operating system can effectively perform this function while using only facilities which are also available to the user then the operating system may be useful for other statistical processing tasks.