ARRAY THEORY IN AN APL ENVIRONMENT

A. Hassitt
L. E. Lyon
I. B. M. Scientific Center
1530 Page Mill Road
Palo Alto, California 94304

Abstract: Trenchard More has proposed an array theory which offers a powerful set of operators and operations on nested arrays. We have written a program called Array Theory/370 (AT/370) which implements these array theoretic operations at the assembler-language level on an IBM/370. We have also written a VS APL auxiliary processor which enables the APL user to access AT/370 and to manipulate nested arrays.

This paper describes the simple but powerful APL-AT/370 interface and illustrates how the interface was used to build an interactive array-theoretic system.

INTRODUCTION

ARRAY THEORY AND AT/370. There has been a considerable interest in recent years in investigating operations on nested arrays (1,2,3). Trenchard More, in particular, has developed an extensive theory (4,5,6,7) that describes a set of operators and operations on nested heterogeneous rectangular arrays. Some of the operations on nested arrays appear, at first sight, to be quite complicated. Experience with APL suggests that actual use of these operations is the fastest and easiest way to comprehend them. It also suggests that the usefulness of the operations can only be evaluated by using them in the solution of practical problems.

We had gained some experience of array theory by modeling it in APL; however, modeling heterogeneous nested arrays in terms of APL's homogeneous simple arrays was so inefficient that it could not serve as a valid test. We decided to implement it in a lower-level language and, after some consideration, chose to use PL/S II (8). PL/S II is a systems-programming language for the IBM/370 used within IBM. This implementation of the theory runs on the IBM/370 and will be referred to as Array Theory/370(AT/370). AT/370 stores, retrieves and performs operations on nested arrays. It has no language, no input-output capabilities and no facility for interactive use; fortunately we can use the services of APL to remedy all of these deficiences.

AT/370 is an experimental program designed for research and experimental purposes. There is no intention at this time of making it part of an IBM product. It is our intention to make the program available within IBM for experiments and research on nested arrays.

DESIGN GOALS. In setting out to implement array theory we decided on several objectives. Since we had chosen More's theory because of its axiomatic approach, a primary objective was to implement the arrays and operations described by More and to not compromise this approach by modifying the design because of the difficulties of implementation. A second objective was to produce a system which could be used on practical problems. This objective required that arrays should be allowed to have reasonably large valence (we allow up to valence 63), depth (we allow up to depth 10, but this could easily be extended) and item count (an item count of up to 2*30 is allowed but in practice a limitation is imposed by the size of the storage area). Valence is the term More uses in place of the APL rank. Depth measures the degree of nesting; an APL scalar has depth zero, an APL non-scalar has depth one, an array whose items are simple arrays has depth two, and so on. We also decided that a reasonably compact representation was necessary, e.g., logical vectors are stored as bit strings. A third objective was to decompose the system into several self-contained parts and to concentrate our efforts on the part of array theory which is well developed, namely the array operations themselves. A final objective was to provide a system which was easy to install, easy to use and easy to experiment with.

VS APL AND AUXILIARY PROCESSORS. AT/370 is for the most part independent of any operating system and can be invoked from assembler language, PL/I, etc. We are concerned in this paper with using AT/370 in conjunction with VS APL under CMS. For those not familiar with VS APL and CMS we summarize the overall features of the system. CMS (9) is a Conversational Monitor System which provides services for developing and running programs in a variety of environments. Assuming that VS APL (10) has been installed, a programmer using CMS can simply type 'VSAPL'. The system responds with

```
      VSAPL
      CLEAR WS
```

and the user is now in the APL environment. The APL user can communicate with external devices such as disks, printers and graphic devices by using an auxiliary processor (AP). Some APs are provided with VS APL and others may be supplied by the user.

As an example, access to a disk file is done by putting the name of the file into an APL variable, for example variable S, and offering to share with a particular AP, typically AP110. The APL statement is:

```
      110 ⎕SVO 'S'
```

Future references to S will get records from the file and assignments to S will put records in the file. As we shall see, a similar mechanism is used to read and write data to an array storage area.

AT/370 AND ARRAY COMMANDS

We have written an array-theory auxiliary processor(ATAP), to interface AT/370 and VS APL. We have made no changes to CMS or to VS APL. The programmer using CMS can invoke the system by simply typing 'AT'; the system responds with

```
      VSAPL
      CLEAR WS
```

VS APL can be used in the usual way and in addition

```
      222 ⎕SVO 'X'
```

can be used to share a variable such as X with ATAP (we are assuming that ATAP is auxiliary processor 222). Any assignment to X will now cause data to be sent to ATAP and any reference to X will read a value set by ATAP.

When ATAP is in use, the programmer has two areas available for storing data, namely the APL workspace and a separate area called the array-storage area (11). We will use the word 'array' to mean an array as specified by More's theory; we will use words like scalar, vector and matrix to refer to APL objects. The array storage area is initially empty. Commands sent to ATAP can cause AT/370 to create arrays, do array operations and to extract array results from the storage area.

ARRAY IDENTIFIERS. Each array in the storage area is known by a non-negative integer. If an array, say array 35, has any items then these items are known as 35 0, 35 1, 35 2, and so on. In general any array or array item can be identified by one or more integers. We preface these integers by an integer specifying the number of integers and refer to the resulting vector as an array identifier, thus 1 35 identifies array 35, and 2 35 0, 2 35 1, 2 35 2, ... identify its items and 3 35 0 0, 3 35 0 1, 3 35 0 2, ... identify the items of the first item of array 35 (array identifier 2 35 0), and so on.

PUTTING VALUES IN ARRAYS. Arrays are created by an APL statement which has the form

```
      X← 0, TYPE, AID
```

where 0 indicates that a put is needed, TYPE is an APL variable described below and AID is an APL variable which gives an array identifier. This statement should be followed by an APL statement which assigns the value of the array to X. For example:

```
      X← 0 0 1 35
      X← 10 0 64 77 123
```

will cause array number 35 to be the five item list 10 0 64 77 123(More uses 'list' in place of APL's use of vector). In this case the TYPE was zero indicating that the type of the array should be the same as the type of the APL variable. The user can force a specific type by using TYPE=1 for boolean, 2 for integer, 3 for real and 4 for complex.

The put statement can be also be used to change the value of any array or existing array item. This simple mechanism can be used to build up nested arrays in the storage area. For example:

```
      X← 0 0   1 7
      X← 10 20 30 40
      X← 0 0   2 7 1
      X← 2 3 ρ'PQRSTU'
      X← 0 4   2 7 3
      X← 54.3  2
```

creates array 7 as a list of integers and then replaces two of its items; the result is that array 7 becomes a list with 10 as item 0, a 2 by 3 array of 'PQRSTU' as item 1, 30 as item 2, and the complex number 54.3R2 as item 3. As the example shows, this method is powerful but not always convenient. Fortunately TYPE=5 offers an alternative method of building the array. The single statement

```
    X← 0 5   1 7
    X← '(4⍴10 (2 3⍴''PQRSTU'') 30 54.3R2)'
```

achieves the same result as the statements given above. When TYPE is 5 the value of X must be a character vector of the form:

```
      ( N ⍴ ITEM1 ITEM2 .... ITEMN)
```

where each item has one of the forms:

<u>TYPE</u>	<u>FOR EXAMPLE</u>
BOOLEAN	○ OR ⊥
INTEGER	¯2345
REAL	34.56
COMPLEX	3R4
CHARACTER	'X'
PHRASE	_FOX
FAULT	⎕ERROR
ARRAY	(3⍴_CATS ⊥ 37.4)

More uses ○ and ⊥ for boolean zero and one. Phrases and faults are motes (that is, scalars, in APL terminology) which hold a string of characters. The parentheses and item count are redundant in some cases; however they are useful in checking for errors in the format of the data. AT/370 parses the character string and translates each item to its internal form, for example 32-bit binary form for integers.

<u>GETTING VALUES FROM ARRAYS</u>. Arrays can be retrieved from the array storage area into the APL workspace by statements of the form:

```
      X← 1, TYPE, AID
      C← X
      V← X
```

TYPE is an APL integer variable as described in the previous section; it gives the type of APL result required. C is an integer vector. Its first item is a return code and the remaining items give the valence, count and type (boolean, integer, real, complex, character or general) of the array. If the first item of C is zero (indicating no error occurred), the next reference of X yields the value of the array. Notice that any array can be represented as an APL character vector by setting TYPE=5 but only simple homogenous arrays can be represented by the other types.

<u>ARRAY OPERATIONS</u>. Arrays can be manipulated by statements of the form:

```
      X← 2, OP, AID, BID
```

or

```
      X← 2, OP, AID, BID, CID
```

where OP is an integer used to specify an operation. The first form is for monadic operations and the second form is for dyadic operations. In each case AID specifies where the result is to go; for example, if 88 is the operation code for PLUS, then

```
      X← 2 88    2 7 3   1 35   3 12 5 9
```

will add array 35 to the item number nine of item number five of array 12 and will put the result in item number three of array 7.

There is another statement which allows a vector of array identifiers and one or more operations or operators. It is used whenever an array-theory operator is used; this occurs in More's generalization of the APL reduction and inner and outer product.

<u>ERRORS</u>. A return code is available after all put, get and array-theory operation statements. In the PLUS example given above, an error would be reported if the array 35 had no value, if array 12 had no sixth item (item number five is the sixth item), if that item had no tenth item or if array 7 had no value or had no fourth item. These errors would correspond to VALUE or INDEX errors in APL.

<u>CURRENT STATUS</u>. The auxiliary processor and a major part of AT/370 had been written and tested. In AT/370 the get command, put command, each operators, all the primitive structural operations and about 40 other operations had been completed. Once the basic routines were written, it was not difficult to implement most of the operations, but a few were quite challenging. The most difficult so far has been dyadic each, which allows items from a left argument to be combined with items from a right argument in a wide variety of ways. The difficulty comes of course from the variety of combinations, variety of operations and variety of arrays which must be allowed for. Dyadic each has been written and tested and we see no problem in any of the other operators or operations.

BUILDING A USER-ORIENTED INTERFACE

The array statements, described above, give the APL program all of the power and flexibility of AT/370, however, they are not particularly convenient to use. We have developed two APL workspaces that use APL functions to build, in the first case, a simple but usable interface and, in the second case, a more complex but more attractive interface.

<u>THE SIMPLE INTERFACE</u>. The first workspace has defined variables such as *∆PLUS* and *∆RESHAPE* that give mnemonic names to the operations. It has a function *∆FINDARRAYS* such that

```
          ∆FINDARRAYS 6 5⍴'ALPHABETA A    B
C    D
```

sets ALPHA to 0, BETA to 1, A to 2, B to 3, C to 4 and D to 5. The user can now think of array zero as having the name ALPHA, array one as having the name BETA, and so. The APL function *∆IS* allows the user to type

```
      (BETA,3) ∆IS  C,∆PLUS,A,5,9
```

for example, instead of the statement

```
      X← 2 88    2 1 3   1 4   3 2 5 9
```

Finally the workspace contains several functions for putting and getting arrays, for example

```
      ALPHA ΔPUTINTEGER  3 4ρι12
```

will set array ALPHA to be a 3-by-4 array of integers.

ARRAY-THEORETIC STATEMENTS. We felt that to properly assess the value of array theory, we would need the convenience and ease of expression which is found in APL itself. The second workspace contains APL functions which allow the use of array-theoretic statements in calculator mode, and the ability to fix (fix in the sense of ⎕FX) array-theoretic functions.

The calculator mode is provided by the APL function ΔAT which reads a line from the terminal, scans it using the syntax described in references (4,7), and executes it. For example:

```
      ΔAT
      ι2 3
 2 3ρ(2ρ0 0) (2ρ0 1) (2ρ0 2) (2ρ1 0) (2ρ1 1)
(2ρ1 2)
      A← _BROWN _JONES _SMITH
      B← 10     20     30
      A _, B
 (3ρ_BROWN _JONES _SMITH) (3ρ10 20 30)
      A :,_ B
 3ρ(2ρ_BROWN 10) (2ρ_JONES 20) (2ρ_SMITH 30)
      ∪(A _, B)
 _BROWN _JONES _SMITH 10 20 30
```

We can terminate the action of ΔAT by entering a blank line. ΔAT creates a function called ΔCODE for each line entered and then executes ΔCODE. We can see the translation of any line by displaying ΔCODE.

```
      ∪(A _, B)
```

translates to

```
     ∇ Δ←ΔCODE
[1]   ΔTMP1 ΔIS  A, ΔPAIR , B
[2]   ΔTMP1 ΔIS ΔUNION, ΔTMP1
[3]   Δ←ΔGET ΔTMP1
     ∇
```

The functions in the workspace check all ATAP return codes and stop if an error occurs. The user can display the value of any array by typing ΔAT and the name of the array to be displayed.

ΔAT accepts data written in More's strand notation (4), for example

```
      A← 10 'ABCD' (3 4 5) o⊥⊥⊥oo⊥
      B← 3 (2 3ρ∘A) 99
```

It translates the first line into a single put statement; this requires some re-formatting of the data to put it into the canonical form required by the put statement. The second line requires several statements since the put statement accepts only constant data.

If we ask for a display of A the result is:

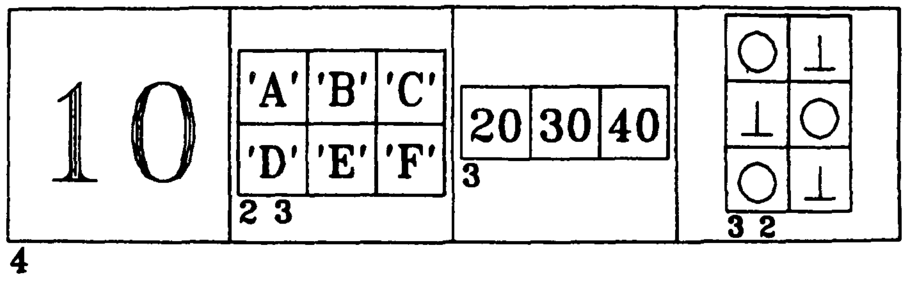
```
      10 (4ρ'ABCD') (3ρ3 4 5) (7ρo⊥⊥⊥oo⊥)
```

This is in the canonical form returned by the get statement with the outermost parentheses removed. Many other forms of display could be written. We are grateful to Peter M. Sih for the following examples of the results of an APL program he has written to display AT/370 arrays on a graphics terminal.

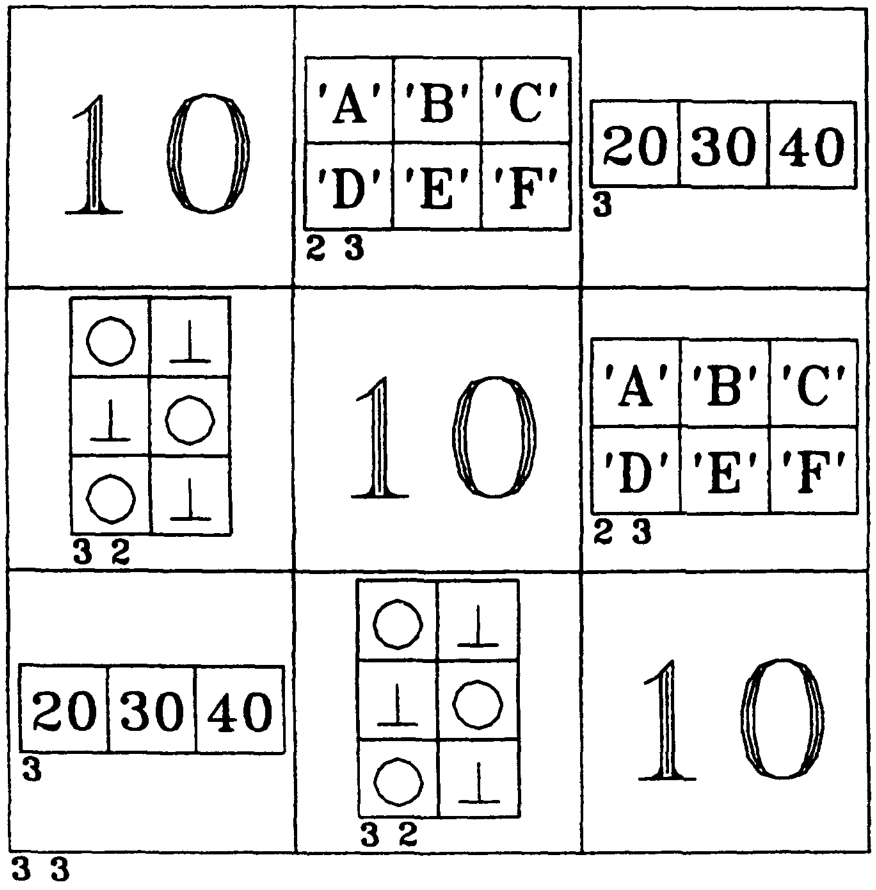
ARRAY-THEORY FUNCTIONS. It should be apparent that once we have APL functions which can translate array-theoretic statements into a function ΔCODE it is then easy to take several such statements and form them into one function. Progressing from a series of statements to an array-theoretic function is a simple matter once we have decided on the form of the function. In choosing a function form we have tried to take the simplest approach,with the

Figure 1. Display of nested arrays using an IBM 3277 Graphics Attachment.

(a) 10 (2 3ρ'ABCDEF') (20 30 40) (3 2ρo⊥⊥o)



(b) 3 3 ρ of array shown in (a).



assumption that future experimentation will lead to an improved form.

The user prepares his array-theoretic function as an APL character matrix, for example:

```
      FM
 Z←A F B
 Z←A + ∪B
```

The function header has the same form as in APL. The statements of the function are array-theoretic statements as described in references (4,7). We allow only two forms of branch statement, namely:

```
     →L
     →L IF S
```

where S is any array statement which produces a boolean result. We need to be able to distinguish between names of functions and names of arrays, so we insist that the names of functions begin with an under-scored letter and the names of variables do not. We have written an APL function called *ΔATFIX* which converts the character matrix into the canonical form of an APL function so that

```
      ⎕FX ΔATFIX FM
F
```

produces an APL function *F* which when executed, issues array statements in order to execute the array-theoretic function specified by *FM*.

CURRENT STATUS. The functions described in this section of the paper have been written and tested. They have been used extensively in the testing of AT/370 where they have proven to be of great value. They seem to meet our objectives of providing an easy-to-use array-theory system.

CONCLUSION

The introduction of nested arrays into APL presents formidable problems in many areas. It raises the question of how arrays and array operations are to be defined, what operations are to be allowed and how are they to be implemented. There is a separate question of whether array-theoretic statements can be designed as a superset of APL statements or whether a new syntax be required. We have chosen to use More's array theory because he has a mechanism for precisely defining all operations in terms of a set of primitive operations. We have implemented the array-theoretic operations in a form which should allow them to be used on practical problems. Since the language aspects of nested arrays are still far from being resolved, our implementation is language independent. APL functions provide a simple language facility which can serve as the basis for future experimentation. It can be objected that the use of an auxilary processor makes the implementation inefficient. It is not difficult to see how this ineffícency can be overcome. Array-theory functions could be transmitted to, stored in and then executed from the array storage area. In that case the AP would only be used for transmitting data and results between the array storage area and APL.

We wish to thank R. J. Creasy for initiating and supporting this work and we are indebted to Trenchard More for many helpful discussions during the course of the project. The storing and retrieving of arrays in AT/370 is done by a program written by L. J. Woodrum; we are grateful to Woodrum for supplying the program.

REFERENCES

(1) Alfonseca, M. and Tavera, M. Extension of APL to Tree-Structured Information. Proceedings of APL76 Conference, page 1.

(2) Gull, W. and Jenkins, M. Recursive Data Structures in APL. C.A.C.M., volume 22, number 2, pages 79-96, February 1979.

(3) Mein, W. Toward a Data Structure Extension to APL. Proceedings of APL76 Conference, page 38.

(4) More, T. A Theory of Arrays with Applications to Databases. I.B.M. Cambridge Scientific Center report G320-2106, September 1975.

(5) More, T. Types and Prototypes in a Theory of Arrays. I.B.M. Cambridge Scientific Center report 320-2112, May 1976.

(6) More, T. On the Composition of Array-Theoretic Operations. I.B.M. Cambridge Scientific Center report 320-2113, May 1976.

(7) We have had extensive discussions with Trenchard More during the course of this work. The implementation reflects many of More's recent developments.

(8) Guide to PL/S II. IBM order number GC28-6794.

(9) IBM Virtual Machine Facility/370: CMS User's Guide. IBM order number GC20-1819.

(10) VS APL, IBM program product number 5748-AP1. See the 'APL Language', IBM order number GC26-3847 and 'VS APL for CMS: Terminal User's Guide', IBM order number SH20-9067.

(11) Note. When VS APL is loaded it is requested to not use all of the available storage. AT/370 gets space needed by issuing a storage request to the CMS operating system.