An Array-theoretic Look Beyond APL2 and Nial

Trenchard More, Jr.
238 Marlboro Road
Sudbury, MA 01776

ABSTRACT

The extended abstract below outlines the main points of a more complete paper in preparation. The purpose of the paper is to trace seven lines of development that have carried array theory from some of its original questions through a succession of intermediate answers to current suggestions for a more coherent theory. The paper explains the algebraic motivations for the theory and how and why the treatment of empty arrays in the Nested Interactive Array Language, Nial, differs from that in APL2. The paper introduces modular indexing, indexing from the end of an axis with negative indices, arrays viewed as tori, arithmetical and Boolean operations on non-numbers and nontruth values, and a purely structural operation underlying reshaping.

EXTENDED ABSTRACT

Version III of array theory built on earlier versions that the author began in 1968 when he joined the APL Design Group. This version, which the IBM Journal of Research and Development published in March 1973 under the title *Axioms and Theorems for a Theory of Arrays*, presented in the framework of an algebra some of the fundamental concepts that led from APL to IBM's APL2 and STSC's APL*PLUS II, such as self-containing scalars, empty arrays having nested structure, strand (nested-vector) notation, the application of operations to nested arrays with scalar operations acting pervasively, pictorial representation of such objects by nested boxes, explicitly defined primitive operators such as *EACH*, and the application of arbitrarily defined operators to any operation. Version IV, developed in 1973-74 and documented in three IBM Cambridge Scientific Center technical reports in 1975–76, simplified the treatment of empty arrays with the introduction of prototypes. APL2 differs substantially from the theory in the implementation of many of these concepts because APL2 maintained compatibility with the original APL and focused on the requirements of a contemporary programming language, while the theory strove to minimize the number of execution-stopping errors, maximize the number of universally valid equations, and preserve in certain array-theoretic expressions the patterns of traditional mathematical formulas.

The goal of array theory is a mathematical model of the ways in which the orthogonal arrangement of material bodies interacts with their hierarchical nesting. The theory grows out of geometrical experience with finite collections of bodies gathered together in orthogonal arrangement. Hierarchies enter this experience because a collection of bodies is again a body as a collection of weights is again a weight and a collection of rulers placed end to end is again a ruler.

Set theory grows out of open sentences expressed in the first-order predicate calculus that describe properties of an object, such as the primeness or evenness of an integer. An open sentence, such as *"N is an integer and N is even"*, is transformed to an abstraction, such as *the set of even integers*, that can be visualized as an aggregation capable of being treated as a whole and therefore as a mathematical object having properties. In treating infinite aggregations as wholes, a concept that according to some mathematicians has never been clear, set theory provides insight into the nature of hierarchical nesting. However, the distinction between the finite and the infinite limits arrays to the representation of finite structures. Linear algebra provides insight into the nature of orthogonal arrangement by using orthogonally arranged collections of coefficients to represent vectors and tensors in abstract spaces. The task of array theory is to express a synthesis of these two insights in a simple algebra.

Arrays, like bodies composed of mass and energy, are complex. Operations on such objects tend to be simpler. It is the relationships between operations that are simple enough to be remembered and communicated. At this level of abstraction, human imagination can exercise its power of design and prediction. From an array-theoretic point of view, a single number is a special case of a corporate database. Any operation applied to one applies to the other. The result of such application, if computation terminates, is always an array subject to further operations, even if the array serves as an error message. Similarly, the result of a physical operation on a body composed of mass and energy is again such a body subject to further operations. If array-theoretic operations and their algebra can be made simple enough, as measured by the brevity and universality with which properties of the operations can be expressed, then it may be possible to both imagine and document, perhaps with an advantage of conciseness, the transformation of assembled data as successive operations on arrays.

The reason for preserving the patterns, at least in form, of traditional mathematical equations and formulas is that such formulas express, for the most simply structured data, relationships that apply almost unchanged to the general case. For example, the addition of matrices behaves like the addition of numbers: the formulas for both look much the same. The relationships are initially learned from the experience of manipulating elementary examples. In acquiring this experience, one learns to recognize a formula as if it were a familiar face. Too great a change in the pattern impedes the transfer of experience to new situations.

The theory suggests that what appears to be the absence of hierarchical and orthogonal structure for an object, such as the absence of axes and the apparent lack of nesting for a number, may in fact be a manifestation of a boundary-case presence of structure. The theory does not view arrays as generalizations of simpler objects but rather

simple objects, including numbers, truth values, characters, and phrases (indivisible strings of characters), as special cases of arrays. To be consistent with this point of view, one must search for formulas that hold for all arrays and explain elementary examples in the context of such formulas.

The concept of number, for example, has evolved to that of any object satisfying certain algebraic laws. The same is true for truth values. A subordinate goal of the theory is to simultaneously weaken these laws as little as possible and define array-theoretic operations so that arrays, including characters and phrases, have almost all the properties of numbers and truth values. The zero, negative numbers, and complex numbers were once thought to be nonsense. Are *"array numbers"* a form of nonsense? With machines for massively parallel computation on the horizon, do we broaden the concept of number to include hierarchical nesting and orthogonal arrangement, so that an array having deeply nested structure is effectively a number because it behaves almost like a number when combined with other arrays by numerical operations, or do we say that computation must stop with an error message when numerical operations encounter characters or phrases or nonconformable arrays?

The salient feature of array theory, perhaps its uniqueness in the study of structures for assembling data, is that the aesthetic sense of algebraic simplicity defines the properties of an array rather than the intuition. As the algebra has become simpler, the array has become more complex. The complexity in turn is difficult to manage without guiding principles at the level of abstraction of an algebra and difficult to express without an increasingly familiar and simple appearance to that algebra.

The foregoing considerations, augmented by discussions with Mike Jenkins and Carl McCrosky at Queen's University in Canada and Ole Franksen, Peter Falster, and Flemming Schmidt at the Technical University of Denmark, and combined with the practical requirements of incorporating the theory within a language concise enough to fit on a personal computer, have widened the divergence of array theory from both set theory and APL. The language that incorporates the evolving theory is called the Nested Interactive Array Language, Nial – an acronym suggested by Franksen. One of the ways to look beyond APL2 and Nial is to trace some of the lines of development that have guided the growth of the theory – lines that originated with questions encountered early in the work, progressed through a succession of answers, and continue to evolve.

Before putting seven of the questions, it is useful to note that the algebraic objectives of the theory require its basic operations to be total and the theory to be closed. Totality means that an operation is defined for every array: application of the operation to an array initiates a computation that halts with the return of a value. Closure means that the value returned is again an array. Given the necessary inclusion of some sort of conditional or branching mechanism, one can of course use the basic operations together with the conditional to define a partial operation that fails to halt and is therefore not defined for certain arrays.

Version III noted a family of operations, including +, for which the monadic operation *f* applied to the pair *A B* returns the same result as the dyadic variant of the operation applied to the array *A* as left argument and the array *B* as right argument. Work with Jenkins on incorporating the theory in Nial and Nial in a small machine required substantial simplification to the syntax of the theory. At the beginning of 1982, he and the author converged to the idea of having the foregoing relationship hold for every array-theoretic operation, with the consequence that every operation name can be prefixed to one argument name or infixed between two argument names. Thus *A f B = f A B* where *"A B"* names the pair *A B* and *f A B = f (A B)*. Equality of arrays means that the arrays are indistinguishable by array-theoretic operations. There is in Version VII, which has been the basis of Nial since 1982, no distinction between monadic and dyadic operations. Association is to the left as in conventional mathematical syntax. Thus *A f B = (A f) B*. The infixing of an operation name means that it is curried with an array name on the left. For example, *2 + 3* is the integer *5* that results from applying the operation *2 +* to *3*.

The paper concerns the lines of development that progress through a succession of answers to the following seven questions:

1. Given the primitive operation *first*, which maps an array to the first item in the array, what array results from applying *first* to an empty array? For example, *first A B = A* and the operation *A first* maps every array to *A*. If the operation *void* maps an array *A* to the empty list that does not hold *A*, then what effect does the operation *first void* have on *A*?
2. What array results from applying an arbitrary operation to each item of an empty array? In other words, if *f* is an operation and *A* is an empty array, then how is the array *EACH f A* related to *A*? According to the IBM Program Product *APL2 Programming: Language Reference*, "An attempt to apply a defined function to each item of an empty array generates a *DOMAIN ERROR*." Errors of this nature make it difficult, if not impossible, to develop a coherent algebra of array-theoretic operations. This second question is closely related to the first because if we knew what was "inside" an empty array, which in Version VII is called the *archetype* of the empty array, we might know how to apply an operation to that archetype and return an empty array of the same size (shape) but with a new archetype.
3. Given the basic operation *pick*, which uses the first item of its argument as an address to pick an array located in the second item of the argument, what is the result of applying the operation *A pick* to an array *B*, where *A* may be any array? This third question amplifies the first because *first* is the same operation as *0 pick list*, where *list* corresponds to APL ravel and lists the items of an array in principal (row-major) order. The natural number *0* indexes the first position on an axis for an array.
4. Given the primitive operation *tell*, which maps a natural number to the list holding the preceding natural numbers, what array results from applying *tell* to an arbitrary array? For example, what is the array *Z* equal to *tell ((2 –3) 4) 'abc' (2.718 (o llol))*, where the pair *2 –3* holds the positive integer *2* followed by the negative integer *–3* and *llol* is a 4-long list of truth values? This fourth question is related to the third because *tell* is the operation that generates arrays of addresses.
5. Given the primitive operations of ordinary numerical arithmetic, such as *sum* or *plus* or + for summation, *prod* or *times* or * for product, *opp* for opposite, and *recip* for reciprocal, what arrays result from the application of these operations to arbitrary arrays? For example, what array results from multiplying the sum of *("Test1" 2 "m.")* and *("Test1" 5 "m.")* by *("Test1" 3 "kg.")*?
6. Given the basic operations of propositional logic, such as *or* for Boolean sum, *and* or *&* for Boolean product, and *not* for Boolean negation, what arrays result from the application of these operations to arbitrary arrays? For example, what array results from the Boolean sum of *not "Job–A" ooll 5* with *"Job–A" olol 5*?
7. What is a good choice for a set of elementary primitive operations and operators for the structural heart of the theory? These functions should be purely structural in the sense that they avoid explicit dependence on Boolean or numerical codes. For example, the basic operation *sublist*, which corresponds to APL compression, determines from the first item of its argument an array of truth values that is used to select items from the second item of its argument. Is there a comparable operation or operator that avoids explicit use of such a Boolean code? Similarly, the basic operation *re*, which corresponds to APL reshaping, determines from the first item of its argument a list of natural numbers that is used to represent the axis lengths for

the size (shape) of the result. Thus *2 3 4 re* is an operation that produces a 2x3x4 array holding items taken cyclically in principal order from its argument. Is there a way to avoid explicit use of such a numerical code?

Version V was outlined at the APL79 Conference in an invited paper entitled *The Nested Rectangular Array as a Model of Data* and a companion paper entitled *Nested Rectangular Arrays for Measures, Addresses, and Paths*. The last section of the latter paper explained how this version differed from Version IV principally in the greater generality given to the recursive generation of addresses. The *numerate* operation in Versions IV and V, which evolved to *tell* in Versions VI and VII, corresponds roughly to APL monadic iota or interval in 0-origin only.

The operation *size* maps any array to the size of the array. Given an array *B*, the array *size B* is a list of natural numbers. The definition of *tell* extends to every list of natural numbers according to the following equation:

$$tell\ size\ B = cart\ EACH\ tell\ size\ B.$$

For example, if *B* is a 2x3x4 array, then *size B* equals *2 3 4* and *tell size B* is the 2x3x4 array shown in Figure 1. The preceding equation is an identity in the sense that it holds for every array *B*. The operation *tell size* returns the same result as the operation *cart EACH tell size* when applied to any array. The two operations are equivalent in the sense of producing the same result for the same argument even though the computational processes may differ. Franksen suggested the double equal sign as a notation for this form of equivalence:

tell 2 3 4

0 0 0	0 0 1	0 0 2	0 0 3	1 0 0	1 0 1	1 0 2	1 0 3
0 1 0	0 1 1	0 1 2	0 1 3	1 1 0	1 1 1	1 1 2	1 1 3
0 2 0	0 2 1	0 2 2	0 2 3	1 2 0	1 2 1	1 2 2	1 2 3

Figure 1. The grid of addresses for a 2x3x4 array.

$$tell\ size == cart\ EACH\ tell\ size. \quad (1)$$

Two operations *f* and *g* are equivalent if and only if (iff) they map any given array to the same array.

$$f == g \ \ iff\ f\,A = g\,A\ \ for\ all\ A \quad (2)$$

Discussions with Paul Penfield in 1976 and early 1977 about the nature of addresses enabled the author to reach in March of that year the following surprising equivalence:

$$tell == cart\ EACH\ tell. \quad (3)$$

This recursive relationship causes *tell* of an array *A* to return an array in which each item has the same nested structure as *A* itself. It remains only to determine what *tell* does to atomic arrays that are not natural numbers.

A variety of conceptually less important improvements resulted in Version VI, which Tony Hassitt, Len Lyon, Bob Creasy, and Mike Jenkins used as the basis for the first implementation of Nial. This version was published as *Notes on the Diagrams, Logic, and Operations of Array Theory* by Tapir in 1981 in *The Second Lerchendal Book* edited by Bjorke and Franksen and entitled *Structures and Operations in Engineering and Management Systems.*

At the APL82 Conference in Heidelberg, Jim Brown and Bob Smith announced that they were using array theory as the basis for some of their respective generalizations of APL to nested arrays. In an invited address to that conference, I said that the theory was based on equations and that I hoped that my colleagues and I would be the first to accept any equation that was in some sense more beautiful than an equation we were using.

Having worked on the theory for fourteen years, I was reasonably confident that the principle equations were as simple as could be attained. But no sooner had I returned to the U.S. and resumed joint work with Jenkins and McCrosky, that it became clear, as a consequence of suggestions from Jenkins and Schmidt and a contradiction I was able to derive from Version VI augmented by the suggestions, that Jean Michel had in 1977 and again in 1979 already proposed to Jenkins and me a better equation. The consequence was an avalanche of simplifications in the treatment of empty arrays that left the remainder of the theory intact.

The simplifications attained in the transition from Version VI to VII were startling, somewhat counterintuitive, but so beautiful that we proceeded on faith. With the support of the Danish Technical Research Council and the Technical University of Denmark, I was able to spend the academic years 1987-89 as a visiting professor with Franksen and Falster drafting the beginnings of a book on the evolution and status of array theory. It has been possible to reconsider the seven questions and lines of development discussed earlier and to reach the following tentative answers, which will be amplified in the presentation of this paper.

1. Version IV assumed on plausible grounds that the operation *first void* was pervasive. This operation was identified with *type*, which mapped any given array to the corresponding typical array of exactly the same structure but having zeros for numbers and blanks for characters, etc. By intertwining type with emptiness, I hoped to over-constrain the theory and thereby expose a contradiction. The problem was that no contradiction occurred. Michel suggested that an alternative choice would be to identify *first void* with the operation *pass*, which maps every array to itself. To every array *A* and every orthogonal arrangement having at least one axis of length zero, there corresponds an empty array having that arrangement and having *A* as archetype. An empty list of twos differs from an empty list of threes. In earlier versions of the theory and in APL2, an empty list of twos and an empty list of threes both equaled the empty list of typical integers, which is the empty list of zeros. Thus the empty arrays of APL2 have differed from those of Nial and the theory since the fall of 1982.
2. The array *EACH f A* always has the same size as the array *A* itself. It suffices therefore to consider empty lists because empty lists can be resized to empty arrays. Let *A* be the empty list *void B*. Then *f* applied to each item of *A* returns the array *void f B*. Thus

$$EACH\ f\ void == void\ f. \quad (4)$$

The consequence of this assumption is that the operator *EACH* distributes through the composition of operations according to the equation

$$EACH\ (f\ g) == EACH\ f\ EACH\ g, \quad (5)$$

which was believed to be true in Version III but found to be invalid in the next three versions for certain empty arrays and certain operations. Much of the effort in these intermediate versions was aimed at recovering this relationship for as many operations as possible because it was essential to the algebra. Michel's suggestion removed the exceptions.
3. The leaves of an array are the atomic arrays, such as numbers and characters, that occur at the deepest levels of nesting. Let the operation tentatively named *"core"* list the integral leaves in an array. Every array has an integral core that can be used as an address. For example, the core of the pair *'ab' (-2 llo 3)* is the pair *-2 3* because the orthogonal arrangements at all levels of nesting are made linear, the nesting is flattened, and the nonintegral leaves are removed. The core of an array without integral leaves is the *Null*, which is the empty list of integer zeros. The operation *A pick* is identified with *core A pick*. Since association is to the left, this is *(core A) pick*.

If the operation *f* is applied to each item of a given nonempty array, then the first item of the result is the same as applying *f* to the first item of the given array. This is an inescapable fact that can be summarized by the identity

$$f\,first == first\ EACH\ f \qquad (6)$$

provided that the identity holds as well for all empty arrays. In particular, the following identity should hold:

$$f\,first\ void == first\ EACH\ f\ void == first\ void\ f. \qquad (7)$$

Thus the operation *first void* must commute with every operation *f*. The operation *pass* that does nothing is the only operation that can commute with every operation. This is another reason why *first void* is *pass*.

The associative law necessarily holds for the composition of operations. This means that in any composition of successively applied operations, attention may by focused *locally* on a short sequence of the operations. For the sake of the algebra, we want the greatest *mobility* in changing the order of the operations within a local view of a composition. This is achieved by *laws of commutation*, such as identities (4), (6), and (7). The significance of the *EACH* operator is that it greatly increases the number of laws of commutation. Having reordered the operations and transformed operations within a local view of a composition, we then want to *simplify* a segment of the composition as is done in (1) according to the identity (3), which is an example of a *law of simplification*. The theory is closed, the basic operations are total, and operators have laws such as the *law of distribution* in (5) because this is the way to achieve the greatest number of laws of association, commutation, and simplification.

By analogy with (6), we want the following law of commutation to hold for every array *A*;

$$f\,(A\ pick) == (A\ pick)\ EACH\ f == A\ pick\ EACH\ f. \qquad (8)$$

Let the identifier "?E" be assigned to the phrase *"?E"*. Then *?E* equals *"?E"*. Suppose *A pick* is applied to the array *B* and returns a fault or error phrase, such as *?E*, because the address *core A* is in some way out of range of suitable addresses for *B*. Then *A pick* should return the same fault when applied to the array *EACH f B* for the same reason. According to (8), any operation *f* applied to *?E* must equal *?E*. But if *f* is the operation *C first*, then *C first ?E* equals *C* for every array *C*. Thus every array equals *?E*, which cannot be the case. Reasoning of this nature shows that *A pick* applied to *B* must return the value *h Q* of some operation *h* applied to some item *Q* in *B* and that *A pick* applied to *EACH f B* should therefore return the value of the same operation *h* applied to *f Q*. Then according to (8) again, *f h Q* must equal *h f Q* for every operation *f* and every array *Q*, because *B* can be so constructed as to hold any particular *Q*. Thus *h* must be *pass* because it commutes with every operation. If (8) is to hold, then for every array *A*, the operation *A pick* applied to *B* must return a particular item of *B* or, if *B* is empty, the archetype of *B* according to the reasoning accompanying identities (6) and (7).

The preceding arguments suggest that as far as addresses are concerned, every array can look like a torus having a particular number of axes and that indexing should be done modulo the length of an axis. If an axis has length *5*, then the negative, zero, and positive multiples of *5* all index the first position on the axis and ..., *-6*, *-1*, *4*, *9*, ... all index the last position on the axis. An address is like the last few wheels of an odometer: the preceding wheels to the left need not be present because they all show zeros. The wheels mentioned explicitly may each show any integer. For example, suppose the array *A* is used as an address and that *core A* equals the pair *10 -9*. Then *A pick* applied to the array *tell 2 3 4* in Figure 1 returns the item *0 1 3* because *0* is prefixed to the address to make up for the missing first index, *10 mod* the length *3* of the second axis of *tell 2 3 4* equals *1*, and *-9 mod* the length *4* of the last axis equals *3*.

Again, as far as addresses are concerned, an array cycles on every axis and has arbitrarily many initial axes all of length *1*. If an address has too many indices, it can be truncated to the correct number of indices, indicated by the valence of the array addressed, by omitting an initial segment of indices. Alternatively, the full address with too many indices can be visualized as addressing an array or torus cycling on the appropriate number of initial axes of length *1*. For example, the operation *17 -31 1 2 8 pick* applied to *tell 2 3 4* returns the array *1 2 0*, which is the item at address *1 2 0* in Figure 1.

4. Since *tell* generates arrays of addresses, it is appropriate to have this operation return a list of negative integers in descending order of magnitude when applied to a negative integer. Thus *tell -3* equals the list *-3 -2 -1*. The array *tell -1 2 -3* is a 1x2x3 array of addresses for the east top right octant of a trivalent array, which in the case of *tell 2 3 4* begins with the item *1 0 1* and ends with the item *1 1 3* in Figure 1. Experience has shown that when options are possible it is often best to have an operation do nothing to its argument. This strategy tends to increase the chances of achieving a law of simplification. The operation *tell* maps every noninteger to itself. The array Z in Question 4 above is a 2x3x4 array in which the first item is *((0 -3) 0) 'abc' (2.718 (o llol))* and every other item has the same structure and nonintegral content. Only the core of each such item changes as one progresses in principal order through *Z*.

5. The associative, commutative, and distributive laws for the algebra of numbers all balance in the sense that every variable occurring on one side of an equation occurs at least once on the other side. By appropriately truncating to the length of the shortest axis, as is done in Nial and preceding versions of the theory, or filling out with faults to the length of the longest axis, as is done in Microsoft EXCEL™, nonconformable arrays can be combined in a reasonable way. Arithmetical and propositional equations have to balance in order to make the structure the same on both sides of an equation. The interesting thing to note is that such balancing also takes care of the problem of combining numbers with nonnumbers in arithmetic and truth values with nontruth values in Boolean logic. Schmidt and Jenkins suggested in 1982 that nonnumbers should propagate out of arithmetical operations.

This same principle can be applied to Boolean operations. Arithmetical operations map truth values, characters, and most if not all phrases to themselves. Boolean operations map numbers, characters, and all phrases to themselves. It appears possible to do dimensional analysis as well. This is one way of satisfying Schmidt's request to have nonnumerical data propagate out with an indication of what happened. Meters plus meters equals meters and meters times kilograms is "kilogram*meters," which is a measure of torque. One possibility is to let phrases spelt with a terminal period be subject to the principles of dimensional analysis. Then *"Test1" 21 "kg*m."* is the result of the following computation that was suggested in Question 5 above.

*("Test1" 2 "m." + "Test1" 5 "m.") * "Test1" 3 "kg."* (9)

6. For Question 6, there is the possibility that *not* applied to *"Job-A" ooll 5* returns *"Job-A" lloo 5*. The Boolean sum of this last result with *"Job-A" olol 5* equals *"Job-A" llol 5*.

7. The paper on Version III noted that sublisting corresponds to the axiom of separation in Zermelo-Fraenkel set theory and that a list-abstraction or separation operator could be defined. If *SIFT* is this operator, then for every operation *f* and every array *A*,

$$SIFT\ f\ A =. EACH\ f\ A\ sublist\ A. \qquad (10)$$

An item *X* in *A* occurs or does not occur in the list *SIFT f A* according as *f X* equals the truth value *1*, namely *Truth*, or not. As the theory shifts to a greater emphasis on operators, it seems best to take *SIFT* as primitive instead of *sublist*.

The operation *re* for reshaping or resizing has always been a puzzle because it appears to be fundamental to APL and array theory and yet seems to be overly complex and dependent on a rather arbitrary list of natural numbers to specify the desired size of the result returned. The theory has the purely structural operations *link* and *cart*, which provide the geometric basis for the addition and multiplication of natural numbers, respectively. Linking lists the items of the items of an array. Figure 1 illustrates *cart*. These two operations have their roots in the axioms of unions and power sets for set theory. Both of these operations affect only the top two levels of an array. The items of the items are like billiard balls to be moved around, replicated, or deleted but not changed.

There is a third important, purely structural operation that has these same properties. It is the operation *pack*, which causes all items of an array to have the same size by replicating along missing initial axes where necessary, truncating to the length of the shortest axis, and then interchanging the top two levels. For example, the pack of a pair of trios is a trio of pairs, and *vice versa*. Let $U, V, ..., Z$ be arbitrary arrays. The operation *pack* maps the array in (11) to that in (12) and the array in (12) to that in (11).

$$(U\ V\ W)\ (X\ Y\ Z) \qquad (11)$$
$$(U\ X)\ (V\ Y)\ (W\ Z) \qquad (12)$$

The underlying concept is one-to-one correspondence: all the first items correspond, all the second items correspond, etc. This concept enables operations such as *sum* and *prod* to be defined recursively:

$$sum == EACH\ sum\ pack \qquad (13)$$
$$prod == EACH\ prod\ pack. \qquad (14)$$

One-to-one correspondence can be captured in a simpler operation that is closely related to *re*. The primitive operation *pin* applied to an array A resizes each of the items in A to that of the first item in A and then interchanges the top two levels in the same way that *pack* does. If all the items of A have the same size, then *pin* and *pack* map A to the same array, as is done for the arrays in lines (11) and (12). The word *pin* is chosen to suggest an old method of searching a database kept on index cards with small holes spaced along the top. A long pin was driven through all the holes in the same position and then lifted, bringing out those cards with a fully enclosing hole around the pin and leaving behind those cards whose holes had been notched open. The first item of *pin A* holds all the first items of the items of A and has the same size as A itself. The size of *pin A* is that of the first item of A. The second item of *pin A* holds all the second items of the items of A, where the second item is taken cyclically from any given item in A; and so on through all the items of the first item of A. Items that correspond by occurring at the same positions in the same principal order are pinned together.