Ziad Ghandour
Jorge Mezei

General Arrays, Operators and Functions

Abstract: This paper discusses "general arrays," i.e., arrays in which the items are either scalars or other arrays. Functions are defined to construct, select from, restructure, and in general manipulate such arrays. Operators are presented as functions whose arguments or values are other functions that can be other than scalar functions. The exposition is shortened and simplified by presenting the material throughout in terms of APL.

Contents

Introduction
Arrays as primitive objects, functions and operators
  General arrays
  Building of arrays
  Selection functions
  Generation of indices
  Operators
  Uniform arrays
  Ravel and unravel
  Logical functions
  Less and combine
  Notation for constants
Choice of axes, coordinates and empty arrays
  The axis operator
  Generalized coordinates
  Empty arrays
Conclusions
References
Appendix

Introduction

In many applications it is convenient to represent the data as "general arrays," that is, arrays in which the items are themselves structured. Consider, for example, the manipulation of files within a data base. Let A be a file in which records consist of project number and project name. Let B be a file where records consist of part number and project number. Typically, one may request, when given a part number P, the numbers and names of the projects that use part P.

One may interpret file A as being a vector whose items (the records) are vectors of two items. In such a record, the first item is a number and the second item is itself a vector of characters. File B is a vector whose items are vectors of two numbers. If the universe of discourse includes arrays as general as these, the answer to a typical inquiry about files may then take the form of a relatively simple expression.

This paper is concerned with the definition of such general arrays and means to evaluate them. The exposition is shortened and simplified by presenting the material throughout in terms of APL. New functions for the construction of, selection from, and other manipulation of general arrays are provided. In addition to removing restrictions and introducing new operators and functions, a detailed treatment of generalized coordinates and empty arrays is given.

With the introduction of general arrays the reasons for restrictions on the domains of some APL primitives disappear. For example, in APL each element of $A\iota B$ must be both scalar and an index to A. Hence, A is restricted to be a vector. If the elements of $A\iota B$ can be nonscalar, A can be permitted to be any APL array. Further, if A and B are taken to be general arrays, $A\iota B$ can be used in applications such as dictionary lookup.

As another illustration, the APL expression $A[I;J]$ contains one semicolon, hence A must be of (fixed) rank two and I and J must be computed independently. If a function ϕ is defined for indexing and a vector V can be formed of two items I and J, these restrictions can be removed by replacing the expression $A[I;J]$ by $A\phi V$. The method of indexing in APL, which has long been recognized as being anomalous, is then reduced to just another primitive function. Further, APL indexing is restricted in that only rectangular cross sections of arrays may be selected. By defining a function $\circ$ and by being able to form an array of vectors (indices) B, one may select an arbitrary collection of items from an array A by using the expression $A\circ B$. The ease of selection is further enhanced by allowing parenthesized expressions to appear on the left of the assignment $\leftarrow$, for example $(A\uparrow B)\leftarrow A\uparrow C$.

More important, with the introduction of general arrays, APL operators (reduction, etc.) no longer need be restricted to scalar functions. For example, to obtain a representation of all points in the first quadrant of the plane with integer coordinates not exceeding M and N, respectively, one may write `(⍳M)∘.,⍳N`, where the outer product operator, denoted by `'∘.'`, is applied to the dyadic function catenate. To obtain vectors of the form `1 2 2 3 3 3` one may write `(⍳N),.ρ⍳N`, where the inner product operator is applied to the dyadic functions catenate and reshape.

Additional operators of general applicability are introduced. With the introduction of the itemwise operator, the evaluation of functions item by item on their array arguments, which had been possible only for scalar primitives in APL, is now possible for all functions. With the introduction of the axis operator, the special APL convention for function indexing is no longer necessary. The axis operator systematically provides this facility, and for a larger collection of functions. The fold operator permits repeated application of functions without requiring program loops.

A theory of arrays is discussed by T. More [1]. General arrays and related topics have also been studied by A. D. Falkoff and K. E. Iverson. Many of the definitions in the initial sections of this paper are adopted, sometimes with modifications, from the above sources. The most fundamental concepts adopted may be summarized as follows.

First, analogously to set theory, where one talks only about sets, in array theory one talks only about arrays. The *individual* in Quine's set theory corresponds to the *scalar* in array theory; the individual is a unit class that contains itself while the scalar is an array that contains itself. Any set that is not an individual has, as members, other sets; similarly, any array that is not a scalar has, as items, other arrays, with the additional feature of an order imposed on those items. For a comprehensive treatment of the relationship between set theory and array theory, see the paper by T. More [2].

Second, implicit use has now been made of functions with functional arguments in the APL operators reduction, scan, inner product and outer product [3]. The concept of an operator makes the above usage explicit. An "operator" may be defined as a function that has an argument(s) or result that is itself a function. The functions may be either primitive or defined, or may themselves be the values of operators.

This paper is organized into two main sections. The first introduces general arrays as primitive objects and the operators and functions defined on them. The appendix lists these operators and functions.

The second section recapitulates the function definitions in the first section and introduces the additional considerations of choice of axes, general coordinates and empty arrays.

For the purpose of facilitating the exposition in this paper, certain concepts are introduced, such as "simple array," "uniform array," "item," "component," "index," "path," etc. However, these concepts are not to be taken as proposals for data objects in a programming language; the only data object is the array.

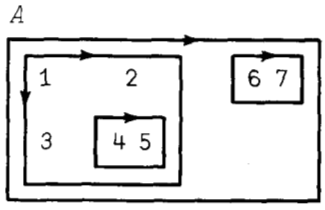
An intuitive graphic notation is employed in the examples. The notation is that of APL\360 output except that each nonscalar array is enclosed in a box and along its sides the axes of the array are represented by arrows. Some familiarity with APL is assumed [3].

Arrays as primitive objects, functions and operators

• *General arrays*

An *array* may be described as an ordered collection of *items*. In APL, arrays are restricted to have scalar items only. The arrays considered in this paper may have any other array as an item.

Example:

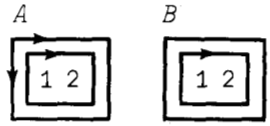


Here, array A is a vector of two items. The first item of A is a matrix of four items. The items of the matrix are the scalars `1`, `2`, `3`, and the vector `4 5`. The second item of A is the vector `6 7`.†

The ordering of items of an array is visualized in terms of a linearly ordered set of *axes*. A *vector* is an array that has one axis only. Each axis of an array has a nonnegative integral *size*, e.g., the vector `1 2 3` has one axis, the size of which is `3`. An *empty array* has size `0` along at least one axis and so has no items.* The *null* ⍬ is the empty vector denoted by `⍳0` in APL. The size of an array is a vector of the sizes of its axes, in their given order.

A *singular array* has exactly one item. A *single* is a singular array that has no axes.

Example:



†In this paper a heavy dot at the end of a sentence should be read as a period.

*Empty arrays are discussed in a later section.

In this example A is a singular array, but not a single; it has two axes and one item, namely the vector 1 2. B is a single.

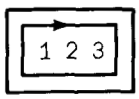
A *scalar* is a single that has itself as its item. Scalars are the only arrays that have themselves as items. A nonempty array is *simple* if and only if every item in the array is a scalar. Thus, APL\360 arrays and scalars are simple arrays restricted in that their items are either all numerals or all nonnumeric characters.

• *Building of arrays*

Essential to the construction of general arrays is the monadic function *enclose*, denoted by <. For any array A, the array $<A$ is a single that has array A as an item.

Example:

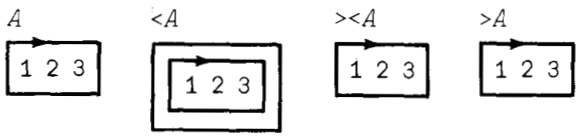
```
<1 2 3
```



Observe that enclose, applied to a scalar, is the scalar itself.

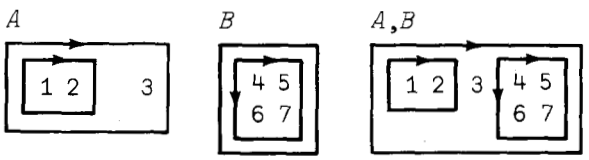
Closely associated with enclose is the monadic function *disclose*, denoted by >. If A is a single, then $>A$ is the item of A, otherwise $>A$ is A. This implies that $><A$ is A and that disclose applied to a scalar is the scalar itself.

Example:



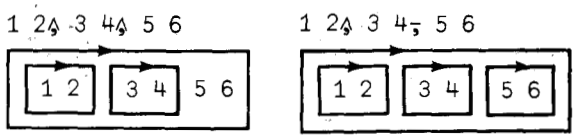
The function *catenate*, denoted by ,, has arguments which are scalar/vectors whose items are general arrays. Consider $Z←A,B$. Z is a vector whose items are the items of A followed by the items of B.

Example:



As a convenience for writing expressions involving enclose and catenate, the functions *link* and *pair* are introduced. Link is a dyadic function, denoted by ⍪, and defined on single/vector arguments such that $A⍪B$ is the same as $(<A),B$. Pair is a dyadic function, denoted by ⍪, and defined on single/vector arguments such that $A⍪B$ is the same as $(<A),<B$.

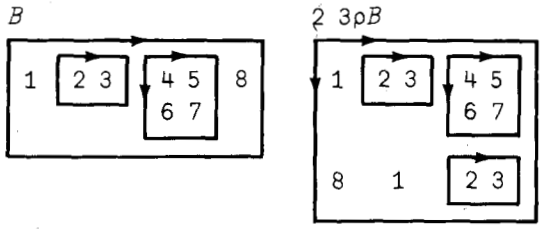
Example:



The function *shape*, denoted by ⍴, applied to a general array has as its value the size of the array.

The dyadic function *reshape*, denoted by ⍴, has for its right argument a general array. Consider $Z←A⍴B$. the size of Z is $,A$ and the item of Z are items of B, obtained in the manner of APL.

Example:

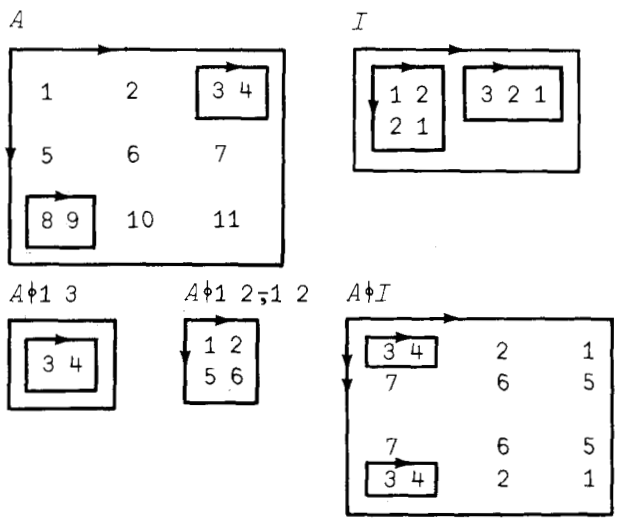


• *Selection functions*

Slice, denoted by ⌷, is a dyadic function for naming sections of general arrays. Thus, for simple arrays, M, I and J, $M⌷I⍪J$ is equivalent to $M[I;J]$.

Consider $Z←A⌷I$, where A is a general array and I is a single/vector having as many items as the number of axes of A, i.e., $(⍴⍴A)$ is $⍴I$. Each item of I must be a simple array. The Nth item of I specifies coordinates along the Nth axis of A. The size of Z is identical to the catenation of the sizes of the items of I.

Example:



The result of the slice function is a name that may occur on the left of assignment, e.g., $(M⌷I⍪J)←M⌷I⍪J$, and similarly for other selection functions.

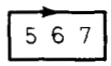
The *index* to an item of an array A is a scalar simple/vector. The items of the index are in one-to-one correspondence with the axes of A and each item specifies the *coordinate* along the corresponding axis. A coordinate along an axis is a positive integer less or equal to the size of the axis.*

Example:

A

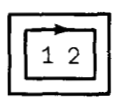
The vector 1 2 is an index to the item 7 of A.

B



The scalar 2 and the vector ,2 are both indices to the item 6 of B.

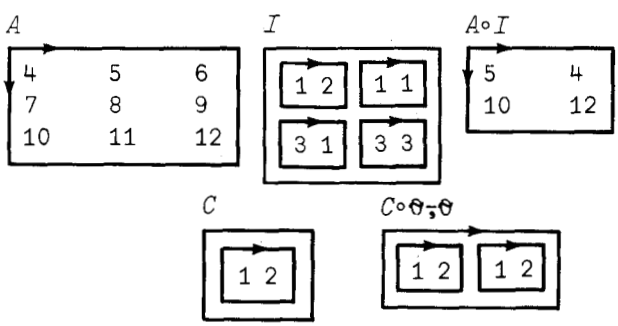
C



The vector ⍬ is an index to the item 1 2 of C (Note C is a single).

Choose, denoted by ∘, is a dyadic function for naming items of arrays. Consider $Z \leftarrow A \circ I$, where I is an array of indices of A. Z is obtained by replacing each item K in I by the item of A to which K is an index.

Example:



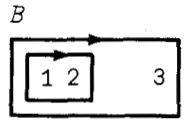
and 5∘<⍬ is 5. Note that ⍴Z is ⍴I.

Given a nonempty nonscalar array A, its *components at level* 1 are the items of A. A component C of A is at level N+1 of A if and only if C is an item of a nonscalar component of A at level N. A has N *levels* if all the components at level N of A are either scalar or empty arrays. A level is a *scalar level* if and only if all components at that level are scalars.

*General coordinates, which may be given from either end of an axis and using different origins are discussed in a later section.

Example:

B



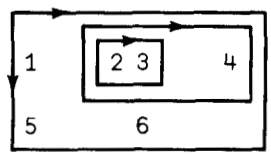
Here, B has 2 levels. At level 1, B has two components: the vector 1 2 and the scalar 3. At level 2, B has two components, namely the scalars 1 and 2.

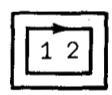
A scalar has no components and no levels. A simple nonempty array A has one level. The components at level 1 of A are the items of A. For example, the vector 1 2 3 has one level. At level 1, it has the scalar componets 1, 2 and 3.

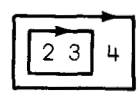
A *path* P to a component C in array A is a single/nonempty vector. If P has one item, that item is an index to C in A. If P has N+1 items and the first N items are a path to a nonscalar component B in A, then the N+1th item of P is an index to C in B.

Example:

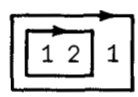
A





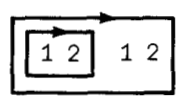
is a path to the component


in A.



is a path to the component

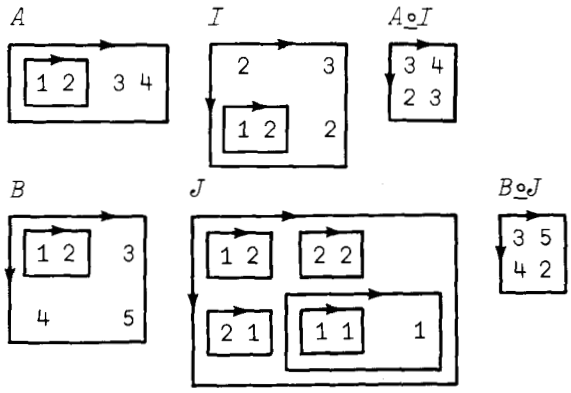

in A.



is a path to the component 3 in A.

Reach, denoted by ⌷, is a dyadic function for naming components of arrays. Consider $Z \leftarrow A \underline{\circ} I$, where I is an array of paths to components of A. Z is obtained by replacing each item K in I by that component of A to which K is a path.

Example:

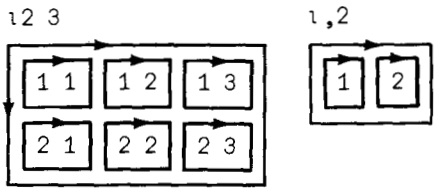


Observe that ρZ is ρI.

• *Generation of indices*

The monadic ⍳ has for its argument a scalar/vector of nonnegative integers. ⍳I is an array of size $,I$. If I has an item 0, then ⍳I is an empty array which is defined in a subsequent section on empty arrays. Otherwise, an item of ⍳I at index J is $(\rho I)\rho J$.

Example:



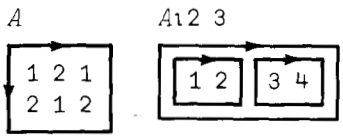
In particular, $(\iota\Theta)$ is $<\Theta$.

Note the identities $A\equiv A\circ\iota\rho A$ and $(\rho\iota I)\equiv\rho I\rho<I$.

For the dyadic functions ⍳ and ⍸, index of a vector is restricted to be a scalar.

The dyadic ⍳ has for it argument general arrays. Consider $Z\leftarrow A\iota B$. Z is obtained by replacing each item K of B by the index of the first item (in principal order) of A which is the same as K. If no item of A is the same as K it is replaced by one plus the index to the last item in A.

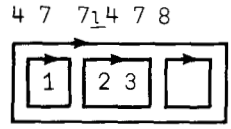
Example:



Also $(5\iota 5)\equiv<\Theta$ and $(5\iota 6)\equiv<\Theta$.

The dyadic function *find*, denoted by ⍸, is defined as follows. Consider $Z\leftarrow A\underline{\iota}B$. Z is obtained by replacing each item K of B by the vector of indices of those items of A which are the same as K. (If no items of A is the same as K, then K is replaced by an empty vector of indices.)

Example:



• *Operators*

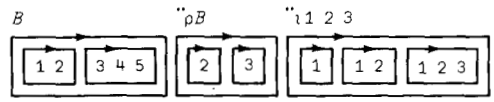
One way to account for the expression $A+.\times B$ is to consider . as a function of arguments + and × whose value is a dyadic function (applied to arguments A and B). An *operator* is a function where at least one of its arguments or its value is a function. *Function* means primitive function or defined function, or the value of an operator. Syntactically, operators behave as functions but with higher precedence. This rule accounts for the interpretation of × as dyadic in the expression above. The monadic operators reduction and scan are written to the left of their (functional) arguments, and the notation for outer product is simplified. Thus one writes $/+B$ instead of the APL expression $+/B$ and similarly, $\backslash+B$ instead of $+\backslash B$ and $A.+B$ instead of $A\circ.+B$.

An important motive for the introduction of general arrays is that it permits the domain of operators to be extended to all (and not only scalar dyadic primitive) functions. Given arrays A, B and functions F, G, H the expression $A\ F.G.H\ B$ is proper and interpreted as $A\ F.(G.H)\ B$. Permitting $A(F.G).HB$ requires the syntactic change that allows expressions that stand for functions to be enclosed in parentheses. (Incidentally, an identifier may be permitted to denote several defined functions with differing numbers of arguments, except for the case of a nil-adic function with value.)

The operators discussed in this paper are restricted to be primitive and to have functions as right arguments and as values. Furthermore, operators are not allowed to be arguments or values of other operators.

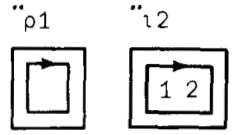
The monadic operator *itemwise*, denoted by ¨, makes explicit a general form of the APL convention according to which scalar primitives are applied to arrays. First, consider $Z\leftarrow\ddot{}\Delta B$, where Δ is a monadic function and B is an array. Z is obtained by replacing each item K of B by ΔK.

Example:



Note that for scalar (in general, single) S, $\ddot{}\Delta S$ is $<\Delta S$.

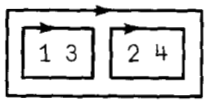
Example:



Next consider $Z \leftarrow A \overset{\cdot\cdot}{\Delta} B$, where Δ is a dyadic function and arrays A and B have the same size. Z is an array of like size. Let K be the item of A at index I, and L be the item of B at index I. Then $K \Delta L$ is the item of Z at index I.

Example:

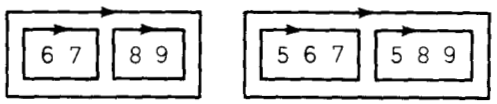
```
1 2 ¨,3 4
```



If either A or B is singular, the APL\360 convention is adopted, i.e., the singular array is reshaped by the size of the other array.

Example:

```
A          5 ¨,A
```

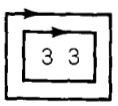


where $(A \overset{\cdot\cdot}{\iota} 7) \equiv A \overset{\cdot\cdot}{\iota} 7\ 7$ and $(A \overset{\cdot\cdot}{\iota} 7\ 7) \equiv 2\ 3$.

If both A and B are singular, the array of smaller rank is reshaped by the size of the other.

Example:

```
2 ¨⍴,3
```



The (generalized) convention concerning scalar functions can now be stated as follows:

Monadic scalar functions Δ are extended to nonscalar arguments B by defining ΔB as $\overset{\cdot\cdot}{\Delta} B$.

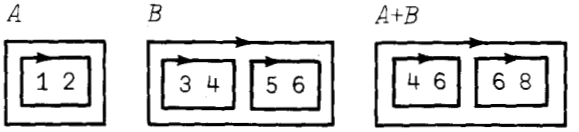
Example:

```
A          -A
```

Dyadic scalar functions Δ are extended to nonscalar arguments A and B by defining $A \Delta B$ as $A \overset{\cdot\cdot}{\Delta} B$, hence in particular either A and B have the same size or at least one of them is singular.

Example:

```
A          B          A+B
```



The monadic operator *reduction*, denoted by $/$, has a dyadic function Δ as argument and a monadic function $/\Delta$ as its value. Intuitively, $/\Delta B$ is $B_1 \Delta B_2 \Delta \ldots \Delta B_{\rho B}$.

More precisely, the argument to $/\Delta$ is a single/vector. The operator is defined by requiring that

$(/\Delta{<}B) \equiv B$

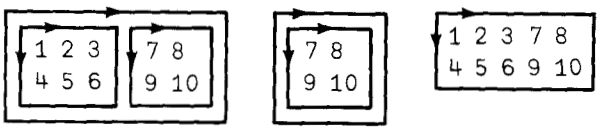
$(/\Delta, {<}B) \equiv B$

and, for vector B of size greater than 1, that

$(/\Delta B) \equiv (>B \circ 1) \Delta / \Delta 1 \downarrow B$.

Example:

```
A          1↓A        /,A
```

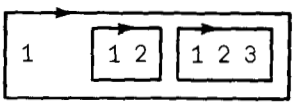


If Δ has an identity I then $/\Delta\Theta$ is I.

The monadic operator *scan*, denoted by $\backslash$, has a dyadic function Δ as argument and a monadic function $\backslash\Delta$ as its value. The argument to $\backslash\Delta$ is a single/vector. Consider $Z \leftarrow \backslash \Delta B$. Z has the size of B and the item of Z at index I is $/\Delta B \circ \iota I$. Taking into account that Θ is an index to a single and $\iota\Theta$ is $<\Theta$ (see the section on empty arrays), then $(\backslash\Delta{<}A) \equiv {<}A$ and $(\backslash\Delta, {<}A) \equiv , {<}A$.

Example:

```
\,1 2 3
```

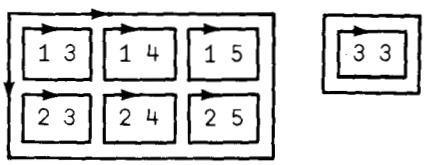


Note that $\backslash\Delta\Theta$ is Θ.

The monadic operator *outer product*, denoted by $\cdot$, has a dyadic function Δ as argument and a dyadic function $\cdot\Delta$ as its value. Consider $Z \leftarrow A \cdot \Delta B$. The size of Z is the size of A catenated with the size of B. Let K be the item of A at index I and L be the item of B at index J. Then $K \Delta L$ is the item of Z at index I, J.

Example:

```
1 2.,3 4 5          2.⍴3
```



The dyadic operator *inner product*, denoted by $\cdot$, has dyadic functions Δ and ∇ as arguments and a dyadic function $\Delta \cdot \nabla$ as its value. The arguments to $\Delta \cdot \nabla$ are singles/vectors. The operator is defined by requiring that $(A \Delta \cdot \nabla B) \equiv / \Delta A \overset{\cdot\cdot}{\nabla} B$ and, in particular, that either A or B have the same size or at least one of them is singular.

Example:

```
1 2 3,.⍴1 2 3 is 1 2 2 3 3 3

2+.⍴1 2 3 is 6 6.
```

The dyadic operator *fold*, denoted by ⍁, has a nonnegative integer left argument N and a monadic function as its value.

If Δ is a monadic function, then $N⍁\Delta B$ denotes what is commonly denoted by $\Delta\ldots\Delta B$ with N occurrences of Δ. In addition,

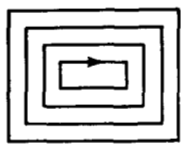
$1⍁\Delta B$ is ΔB

and

$0⍁\Delta B$ is B.

Example:

3⍁<⍬



If Δ is a dyadic function, then $N⍁\Delta B$ denotes what is commonly denoted by $B\Delta B\Delta\ldots\Delta B$ with N occurrences of B, that is, $N⍁\Delta B$ is $/\Delta N\rho<B$. In addition, $1⍁\Delta B$ is B and, if Δ has an identity I, then $0⍁\Delta B$ is I. For example, 3⍁+.×B is B+.×B+.×B.

The monadic operator *limit*, denoted by ⍁, is defined by requiring that $(⍁\Delta B) \equiv K⍁\Delta B$, where Δ is a function and K the smallest nonnegative integer for which $K⍁\Delta B \equiv (1+K)⍁\Delta B$.

For example, (⍁>N⍁<⍬) ≡ ⍬.

The dyadic operator *axis*, denoted by :, has as left argument a scalar/single vector of axis numbers (an axis number is a coordinate for the ordered collection of axes). Intuitively, the axis operator selects those axes along which its functional argument is to be applied. Thus one writes

S:ΔB instead of $\Delta[S]B$ and

$A(S:\Delta)B$ instead of $A\Delta[S]B$.

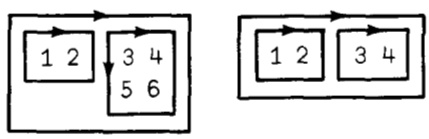
The axis operator will be discussed in more detail in a later section.

• *Uniform arrays*

Level N of an array A is uniform if all components of A at level N have the same size. An array is *uniform to level N* if its initial N levels are uniform. An array is *uniform* if and only if it is uniform at all levels.

Example:

A B



A is uniform at level 2 (not at level 1), whereas B is uniform.

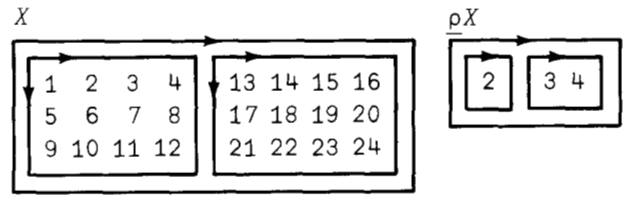
The phrase "axes of B at level N," where B is uniform at level N, refers to the axes of any component of B at level N.

The *cipher*, denoted by ⍰, is a scalar that is neither a number nor a character.

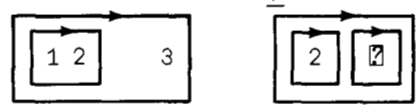
The monadic function *form*, denoted by ⍴, is defined as follows. Consider $Z\leftarrow⍴B$. Z is a vector whose first item is ρB and whose N+1th item I is the size of a component of B at level N if B is uniform at level N. Otherwise I is ,⍰. If B has a scalar level, its scalar level is necessarily uniform and the size of a component at that level is necessarily ⍬. To avoid this redundancy, Z will have no item for the scalar level of B. Thus Z has as many items as there are levels for B.

Example:

X ρX



Y ρY



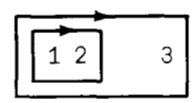
⍴B is called the *structure* of B. The structure of a scalar is an empty vector.

The notion of *principal order* on items of simple arrays as defined in APL is generalized to items of general arrays. Principal order is then extended to components of arrays in the following manner.

The components at level N+1 of A follow the components at level N of A. Given the components I and J at level N such that J follows I in principal order, the items of J follow the items of I with each collection of items in principal order.

Example:

A



The components of A in principal order are, from left to right:

3 1 2

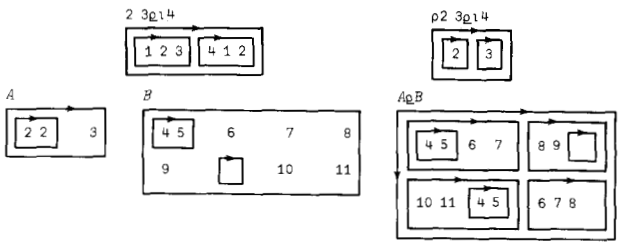
The dyadic function *reform*, denoted by ⍴, is defined as follows. Consider $Z\leftarrow A⍴B$, where A is a single/vector whose items are scalars/simple vectors of nonnegative integers. Intuitively, if N is the number of items of A, and N>1, then Z is an array, uniform to level N−1. The

ravel of the first item of A becomes the size of Z, the ravel of the Kth item of A becomes the size of components of Z at level $K-1$ for $K>1$, that is,

$((\rho,A)\uparrow\underline{\rho}Z)\equiv{}^{\bullet\bullet},A.$

The components of Z at level N, in principal order, are the items of B in principal order, used repeatedly if necessary.

Example:

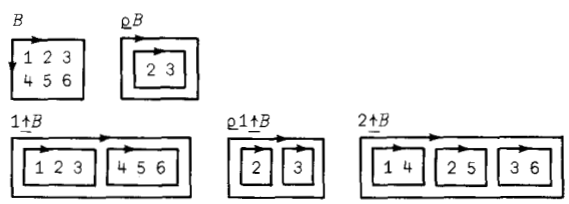


Note that $2\underline{\rho}X$ is $2\rho X$ and $(<2\ 3)\underline{\rho}X$ is $2\ 3\rho X$.

The following structure-modifying functions are especially useful in connection with uniform arrays.

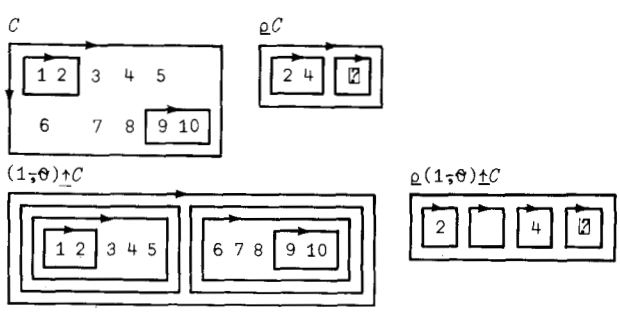
The dyadic function *raise*, denoted by $\underline{\uparrow}$, is defined as follows: Consider $Z\leftarrow A\underline{\uparrow}B$, where A is a single/vector, each item of which is a scalar/simple vector of axis numbers of B. An axis number of B may appear only once in the items of A. Intuitively, if the first item of A is P and I is any proper index of P, then the axis $P\circ I$ of B becomes the Ith axis of Z. Similarly, for $K>1$, if the Kth item of A is P and I is any proper index of P, then the axis $P\circ I$ of B becomes the Ith axis at the $K-1$th level of Z. The remaining axes of B, in their original order, appear at level N of Z, where N is the number of items of A. The Jth level of B becomes the $J+N$th level of Z.

Example:



Also, $(<1\ 2\underline{\uparrow}B)\equiv B$ and $((<\Theta)\underline{\uparrow}B)\equiv <B$.

Further,



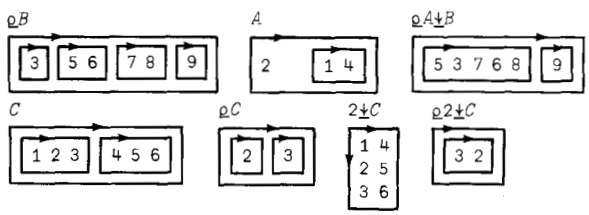
If $(\underline{\rho}D)\equiv 2\ 3\ 4\ 5\_6$, then

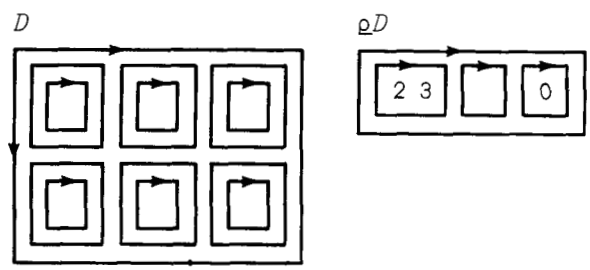
$(\underline{\rho}(<3\ 1)\underline{\uparrow}D)\equiv 4\ 2\_3\ 5\_6$

If A is empty, $A\underline{\uparrow}B$ is B.

The dyadic function *lower*, denoted by $\underline{\downarrow}$, is defined as follows. Consider $Z\leftarrow A\underline{\downarrow}B$, where A is a single/vector, each item of which is a scalar/simple vector of axis numbers of B. If A has N items, B must be uniform to level N. Intuitively, B is "coalesced" to its level N, i.e., the axes of B and the axes of the first N levels of B become the axes of Z. For $J>N$ the Jth level of B becomes the $J-N$ level of Z. The size of Z is determined as follows: let P be the Kth item of A and R be the Kth item of $\underline{\rho}B$. $,P$ and R must have the same size. For any proper index I, $R\circ I$ is the $P\circ I$th item of the size Z.

Example:



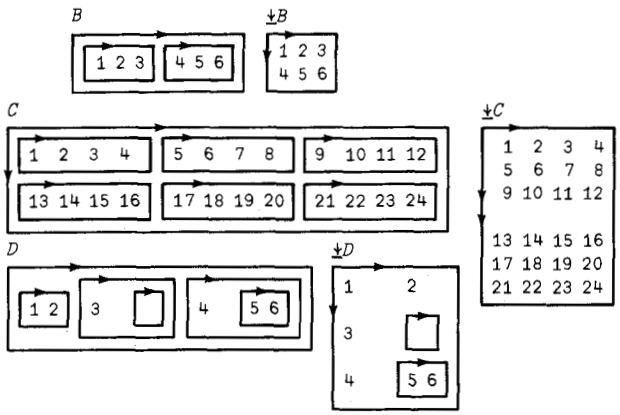


$(3\ 1\ \bar{}\ \Theta)\underline{\downarrow}D$ is $3\ 0\ 2\rho 0$.

If A is empty, $A\underline{\downarrow}B$ is B. Note that $(A\underline{\downarrow}A\underline{\uparrow}B)\equiv B$, and $(A\underline{\uparrow}A\underline{\downarrow}B)\equiv B$.

The monadic function *collapse*, denoted by $\underline{\downarrow}$, is defined as follows. Consider $Z\leftarrow\underline{\downarrow}B$. Let M be the maximal level number such that B is uniform to level M and level M is not a scalar level. Intuitively, B is "coalesced" to its level M. The sequence of axes of Z is the same as the sequence of axes of B, followed by the sequence of axes of B at level 1 through level M. The components at level $M+1$ of B become the items of Z.

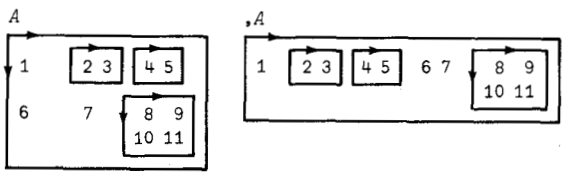
Example:



• *Ravel and unravel*

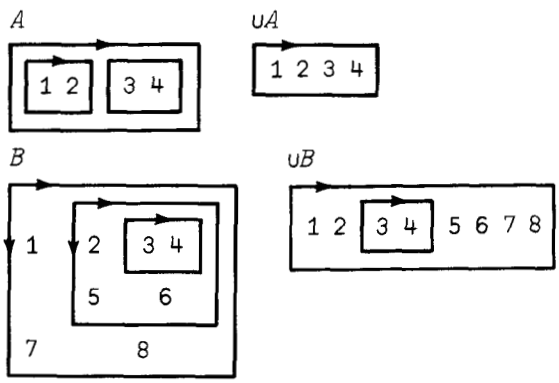
The monadic function *ravel*, denoted by ,, has as argument any general array. ,A is the vector of items of A in principal order.

Example:



The monadic function *unravel*, denoted by ∪, is analogous to set sum in set theory. ∪A is the vector formed by catenating the ravels of the items of A in principal order, i.e., ∪A is a vector of the items of the items of A.

Example:

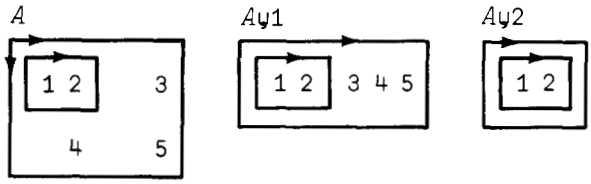


Precisely, ∪A is /,[⍤1],,A

• *Further selection functions*

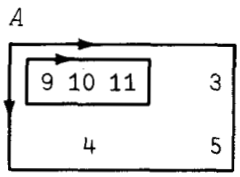
The dyadic function *level*, denoted by ⍷, is defined as follows: Consider Z←A⍷B, where B is a level number of A. Z is the vector of all components of A at level B in principal order.

Example:



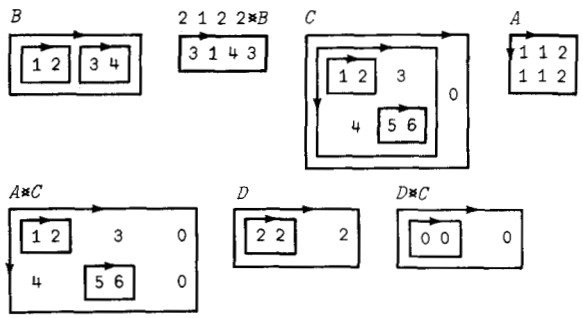
A⍷B is a name that can occur to the left of assignment. For example, after executing

```
(A⍷2)←9 10 11
```



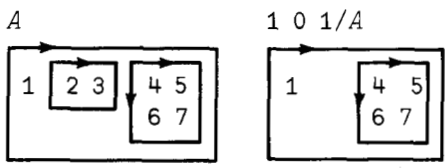
The dyadic function *mesh*, denoted by ⍀, is a selection function defined as follows: Consider Z←A⍀B, where B is a single/vector and A is a general array. Z is obtained by replacing the scalar components of A by the items of the items of B. The scalar components of A of value K are replaced by items of >(,B)∘.=K. This is carried out in principal order and the items of >(,B)∘.=K used repeatedly if necessary. The cases where A is empty or ,B has no Kth item or the Kth item of ,B is empty are discussed subsequently in the section on empty arrays.

Example:



The dyadic function *compress* has for its right argument a general single/vector.

Example:



and 1 0 0/A is ,1.

The result of the compress function is a name that may occur on the left of assignment.

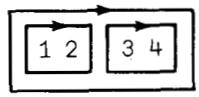
The dyadic function *expand* has for its right argument a general single/vector preserving the identity that A/A\B

is ,B. Hence, as in APL, the following relation must hold:

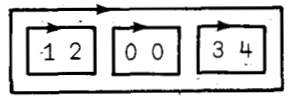
(/+A) = /×ρB.

Examples:

B



1 0 1\B



(1\5) ≡ ,5

(1\,5) ≡ ,5.

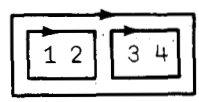
Note that (A\B)≡,(1+~A)⌿<B.

Notice that the item of the result which corresponds to the 0 item of the left argument has the same structure as an item of B, assuming B is uniform, and has scalar components 0. The cases where B is not uniform or is empty are defined later on in the section on empty arrays.

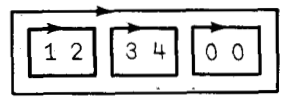
The dyadic function *take* has for its right argument a general array.

Example:

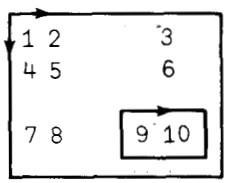
B



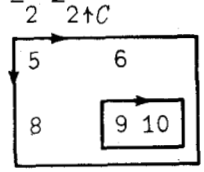
3↑B



C



¯2 ¯2↑C



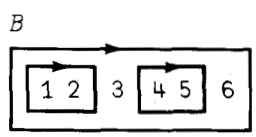
Notice that whenever an item of the left argument exceeds the size of the corresponding axis of the right argument, the result contains items that have the same structure as an item of B, assuming B is uniform, and have scalar components 0. Observe that (ρA↑B)≡|A.

Again, as in compress, the result of the take function is a name that may occur on the left of assignment.

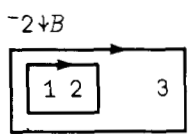
The dyadic function *drop* has for its argument a general array.

Example:

B



¯2↓B



5↓B is an empty array.

Observe that ρA↓B is 0⌈(ρB)−|A.

• *Logical functions*

Arrays A and B are *identical* if and only if

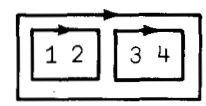
1. either they are the same scalar,
2. or they are empty arrays of identical structure,
3. or they have identical size and have identical items at the same indices.

The dyadic function *identical to*, denoted by ≡, has value 1 if its arguments are identical, 0 otherwise. For example, 3 4≡3 4 is equal to 1, (1 2ρ3 4)≡3 4 is equal to 0, and 1 2≡3 4 is equal to 0.

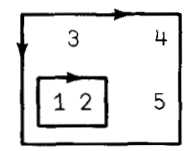
The dyadic function *membership*, denoted ∈, has for its arguments general arrays. A∈B is a simple logical array such that ρA∈B is ρA.

Example:

A



B



(A∈B)≡1 0.

The dyadic function *item of*, denoted by ⊆, has value 1 if its left argument is an item of the right argument, and 0 otherwise. For example, (1 2⊆⍳4)≡0 and (1 2⊆3, <1 2)≡1.

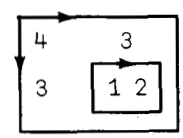
Note that (A∈B)≡A.⊆<B and (A⊆B)≡(<A)∈B.

• *Less and combine*

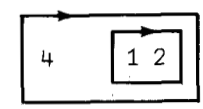
The dyadic function *less*, denoted by ~, has as its value a vector whose items are the items of the left argument that are not items of the right argument, in principal order. Note that (A~B)≡(,~A∈B)/,A.

Example:

A



A~3 5

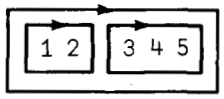


The monadic function *combine*, denoted by ⊗, is defined as follows.

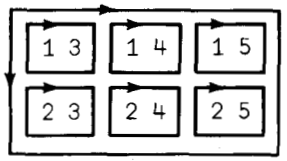
Consider Z←⊗B. The size of Z is the catenation of the sizes of items of B in principal order. The size of each item of Z is the size of B. Intuitively, if ravel of B has N items and J_K is the item at index I_K of the Kth item of ravel of B, for K=1,...,N then (ρB)ρJ_1,...,J_N is the item of Z at index I_1,...,I_N.

Example:

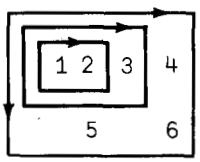
B



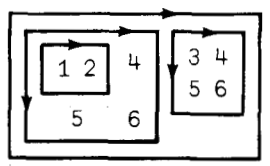
⊗*B*



C



⊗*C*



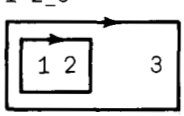

```
(⊗<1 2)=1 2,
(⊗1 2)=<1 2.
```

Precisely, $(\otimes B)\equiv(<\rho B).\rho/.,^{\cdots\cdots}<,B$.

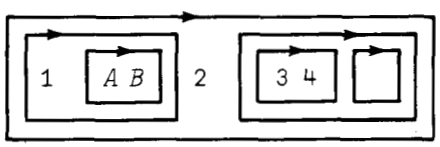
• *Notation for constants*

The input-output notation for constant vectors whose components are either scalar, null or vectors of size greater than one, is defined as follows: Items of a vector of N levels, $N\geq 1$, are separated by $N-1$ underscores.

```
1 2_3
```



Also, `(1_'AB'__2__3 4_⍬)≡(1⍪'AB')⌽2⍪3 4⍪⍬`



Since the underscore does not appear at the right end of such a constant, a typed line ending in underscore may be interpreted as continuation of the expression to the next line. For example,

```
1 2_
3 4 5_
_6_
_7 8
```

is identical to `1 2_3 4 5__6__7 8`.

Choice of axes, coordinates and empty arrays

• *The axis operator*

Some functions are defined for vector arguments and extended to arrays by selection of an axis along which they are applied. In APL the function symbol may be followed be a bracketed expression whose value is an axis number. The *axis operator*, denoted by `:`, performs the same task as the above function indexing mechanism. The left argument of `:` is a single/vector of distinct axis numbers. The right argument of `:` is a function. Intuitively, the right argument is applied along the axes specified by the left argument. In the remainder of this section, the `:` operator is defined for each function that can be the right argument of `:` (with the aid of raise $\uparrow$ and lower $\downarrow$).

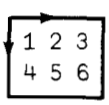
Catenate

Let S be a singular array whose item is an axis number of either array A or array B, then:

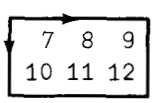
1. If $(\rho\rho A)\equiv\rho\rho B$ then $(A(S:,)B)\equiv S\downarrow(S\uparrow A),S\uparrow B$
2. If $(\rho\rho A)\equiv 1+\rho\rho B$ then $(A(S:,)B)\equiv S\downarrow(S\uparrow A),<B$
3. If $(1+\rho\rho A)\equiv\rho\rho B$ then $(A(S:,)B)\equiv S\downarrow(<A),S\uparrow B$.

Example:

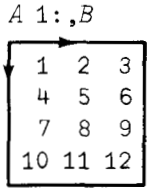
A



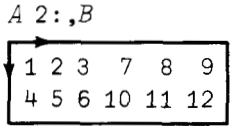
B



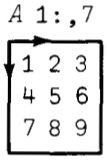
A `1:,`*B*



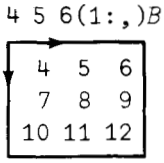
A `2:,`*B*



A `1:,7 8 9`



`4 5 6(1:,)`*B*



The item of S may be a noninteger. In this case A and B must either have the same size, or else one of them is singular and the singular argument is reshaped by the size of the other. The item of S must be equal to an axis number of the resulting arrays plus or minus a number in the open interval `(0,1)`. Then $(A(S:,)B)\equiv(\lceil S)\downarrow A,B$.

Example:

```
A
 1  2  3  4
 5  6  7  8
 9 10 11 12

B
13 14 15 16
17 18 19 20
21 22 23 24

A 0.5:,B
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24

A 1.2:,B
 1  2  3  4
13 14 15 16

 5  6  7  8
17 18 19 20

 9 10 11 12
21 22 23 24

A 2.1:,B
 1 13
 2 14
 3 15
 4 16

 5 17
 6 18
 7 19
 8 20

 9 21
10 22
11 23
12 24

A 0.5:,1 1ρ13
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 13 13 13
13 13 13 13
13 13 13 13
```

Slice

$$(A(S:\phi)B)\equiv A\phi V$$

where $V\leftarrow{}^{\cdot\cdot}\iota\rho A$ and $(V\circ S)\leftarrow B$, i.e., B specifies coordinates along axes S of A. Each axis of A *not* specified by S is taken in its entirety.

Example:

```
A
1 2 3
4 5 6

A 2:φ<1 3
1 3
4 6
```

Choose

$(A(S:\circ)B)\equiv((<S)\uparrow A)\circ B,$

i.e., each item of B specifies coordinates along axes S of A that have been raised.

Example:

```
A
1 2 3
4 5 6

A 2:∘1 3
┌───┬───┐
│1 4│3 6│
└───┴───┘
```

Reduction

Let S be singular. Then

$(S:/\Delta B)\equiv{}^{\cdot\cdot}/\Delta(<(\iota\rho\rho B)\sim S)\uparrow B.$

Example:

```
B
2 3 4
5 6 7

1:/ρB
┌───┬─────┬───────┐
│5 5│6 6 6│7 7 7 7│
└───┴─────┴───────┘
```

Scan

Let S be singular. Then $(S:\backslash\Delta B)\equiv T\downarrow{}^{\cdot\cdot}\backslash\Delta T\uparrow B,$ where $T\equiv<(\iota\rho\rho B)\sim S.$

Example:

```
B
1 2 3
4 5 6

1:\-B
 1  2  3
-3 -3 -3

2:\-B
 1 ¯1  2
 4 ¯1  5
```

Inner product

The case of single/vector arguments has been previously discussed. Otherwise, let I be an axis number of A and J be an axis number of B. Then

$(A(I,J):\Delta.\nabla B)$
$\equiv((<(\iota\rho\rho A)\sim I)\uparrow A).\Delta.\nabla(<(\iota\rho\rho B)\sim J)\uparrow B.$

If A or B or both are single than I or J or both, respectively, must be Θ. Anagolous to the APL\360 convention, if I is $(0\neq\rho\rho A)\uparrow\rho\rho A$ and J is $(0\neq\rho\rho B)\uparrow 1$ then both I and J can be simultaneously elided.

Example:

```
A
1 2 3
4 5 6

B
7  8
9 10

A 1 2:+.×B
39  49
54  68
69  87
```

```
((1 1ρ2)+.×3 4 5) ≡ ,24

(5+.×2 3ρι6) ≡ 25 35 45

((1 1ρ5)+.×<2 2ρι4) ≡ ,<2 2ρ5 10 15 20.
```

Ravel

$(S:,B)\equiv,(<S)\uparrow B.$

Example:

```
A
1 2 3
4 5 6

2:,A
┌───┬───┬───┐
│1 4│2 5│3 6│
└───┴───┴───┘

B
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
```

```
(1 3: ,B)≡1 5 9_2 6 10_3 7 11_4 8 12
_13 17 21_14 18 22_15 19 23_16 20 24.
```

Mesh

Consider $Z\leftarrow A(S:\ast B)$, where:

1. A is simple,
2. $(\rho\rho A)\equiv\rho,S$,
3. The items of B have the same rank,
4. S selects axes of items of B.

Consider each axis of items of B not selected by S. All items of B must have the same size along that axis.

Then $Z\equiv(<S)\downarrow A\ast(<<S).\uparrow B$.

The "meshing" is done along the axes S of the items of B.

Example:

```
B
┌─────┬─────┐
│1 2  │5 6  7│
│3 4  │8 9 10│
└─────┴─────┘
```

```
2 1 2 1 2(2:*)B
┌──────────┐
│5 1 6 2  7│
│8 3 9 4 10│
└──────────┘
```

```
C
┌─────┬─────┐
│1 2  │7  8 │
│3 4  │9 10 │
│5 6  │     │
└─────┴─────┘
```

```
2 1 2 1 2(1:*)C
┌─────┐
│7  8 │
│1  2 │
│9 10 │
│3  4 │
│7  8 │
└─────┘
```

Compress

Let S be singular. Then $(A(S:/)B)\equiv S\downarrow A/S\uparrow B$.

Example:

```
B
┌─────┐
│1 2  │
│3 4  │
│5 6  │
└─────┘
```

```
1 0 1(1:/)B
┌─────┐
│1 2  │
│5 6  │
└─────┘
```

Expand

Let S be singular. Then $(A(S:\backslash)B)\equiv S\downarrow A\backslash S\uparrow B$.

Example:

```
B
┌─────┐
│1 2 3│
│4 5 6│
└─────┘
```

```
1 0 1(1:\)B
┌─────┐
│1 2 3│
│0 0 0│
│4 5 6│
└─────┘
```

Take

$(A(S:\uparrow B)\equiv(<S)\downarrow A\uparrow(<S)\uparrow B$.

Example:

```
B
┌──────────┐
│ 1  2  3  4│
│ 5  6  7  8│
│           │
│ 9 10 11 12│
│13 14 15 16│
│           │
│17 18 19 20│
│21 22 23 24│
└──────────┘
```

```
¯2 3(1 3:↑)B
┌──────────┐
│ 9 10 11  │
│13 14 15  │
│          │
│17 18 19  │
│21 22 23  │
└──────────┘
```

```
C
┌─────┐
│1 2 3│
│4 5 6│
└─────┘
```

```
2(2:↑)C
┌─────┐
│1 2  │
│4 5  │
└─────┘
```

Drop

$(A(S:\downarrow B)\equiv(<S)\downarrow A\downarrow(<S)\uparrow B$.

Example:

```
B
┌──────────┐
│ 1  2  3  4│
│ 5  6  7  8│
│           │
│ 9 10 11 12│
│13 14 15 16│
│           │
│17 18 19 20│
│21 22 23 24│
└──────────┘
```

```
1 ¯1(1 3:↓)B
┌──────────┐
│ 9 10 11  │
│13 14 15  │
│          │
│17 18 19  │
│21 22 23  │
└──────────┘
```

```
C
┌─────┐
│1 2 3│
│4 5 6│
└─────┘
```

```
1(2:↓)C
┌─────┐
│2 3  │
│5 6  │
└─────┘
```

• *Generalized coordinates*

Coordinates along an axis may be given from either end. For an axis of size S, the coordinates I and $I-S$ are equivalent, independently of index origin. For example, in index origin 0,

```
(6 7 8 9 10∘.2)≡6 7 8 9 10∘.¯3,
```

and in index origin 1,

```
(6 7 8 9 10∘.3)≡6 7 8 9 10∘.¯2,
```

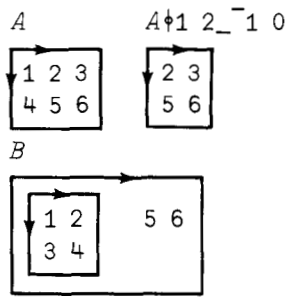
all four expressions denoting 8.

Indices and hence paths may contain general coordinates. Accordingly, the domains of functions and operators are described below, all examples being given in index origin 1.

Slice, choose and reach

These three functions are extended to right arguments with general indices.

Example:



Monadic ⍳

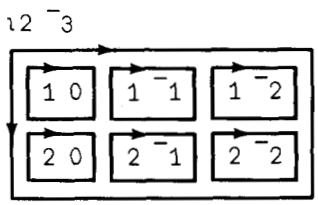
For negative integers B:

$(\iota B)\equiv B+\phi\iota|B,$

i.e., ιB is a vector of $|B$ items. The items are successively smaller integers starting with the integer one less than the index origin. For example, $(\iota\ {}^{-}3)\equiv 0\ {}^{-}1\ {}^{-}2$.

A further extension is for B a scalar/vector of integers, $(\iota B)\equiv\otimes{}^{\cdot\cdot}\iota B$.

Example:



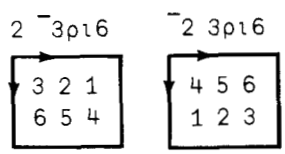
It is in general no longer the case that the item of ιB at index J is J.

However, $(\rho\iota B)\equiv|B$ and $(\iota B)\equiv((\iota B)\circ\iota B)\circ\iota B$.

Reshape

The reshape function is extended to a left argument that is a scalar/vector of integers, and reshape is then defined by the identity $(A\rho B)\equiv((|A)\rho B)\circ\iota A$.

Example:



It should be noted that, in general, $(\rho A\rho B)\equiv|,A$.

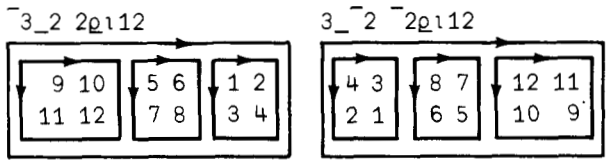
Reform

The reform function is extended to a left argument that is a single/vector whose items are scalars/vectors of integers. The reform function is then defined by the identity

$(A\underline{\rho}B)\equiv((T=T)\neq<\iota\rho\cup T)\uparrow(\cup A)\rho B,$

where $T\equiv{}^{-}1\downarrow A$.

Example:



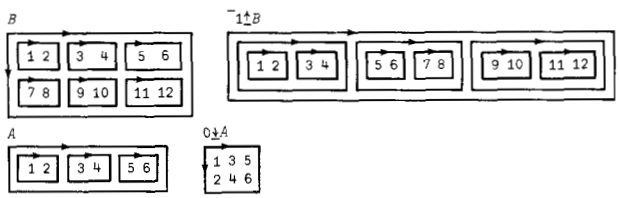
Note that, if A has N items, $A\underline{\rho}B$ is uniform to level N and the following relation holds:

$(N\uparrow\rho A\underline{\rho}B)\equiv 1^{\cdot\cdot},A.$

Raise and lower

The raise and lower functions are extended to left arguments with general axis numbers.

Example:



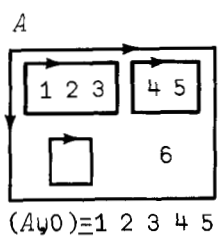
Note that $(0\underline{\downarrow}A)\equiv 2\underline{\downarrow}A$.

Note also that level numbers, like axis numbers, are a special case of coordinates.

Level

This function is extended to right arguments with general level numbers.

Example:

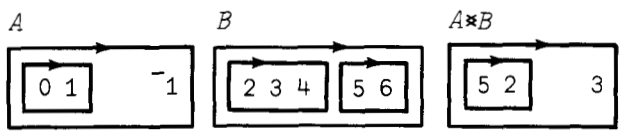


$(A\underline{\downarrow}0)\equiv 1\ 2\ 3\ 4\ 5$

Mesh

The mesh function is extended to left arguments whose scalar components are general indices.

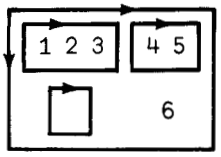
Example:



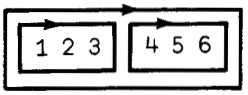
And finally the axis operator is extended to left arguments with general axis numbers.

Example:

B



¯1:/,*B*



• *Empty arrays*

In APL, empty arrays may have different sizes and two empty arrays are identical if and only if they have the same size. In general, empty arrays may have different structures and two empty arrays are identical if and only if they have the same structure. An empty array is completely specified by its structure. The structure of an empty array is a vector whose items are scalars/vectors of nonnegative integers. The first item is the size of the empty array and the last item may not be 0. All empty arrays are defined to be *uniform*, i.e., their structure has no scalar component cipher. An empty array is *simple* if and only if its structure is singular. The simple empty arrays are the APL empty arrays. An empty array has no items, no components, and no levels. The remainder of this section deals with cases where the domain or range of functions are empty arrays.

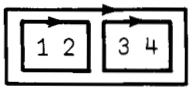
Two cases arise in defining function *reshape*. Consider *Z*←*A*ρ*B*.

Case 1: A has a zero item.

Z is an empty array. In general, the structure of *Z* is obtained by replacing the first item of the structure (⍴*B*) of *B* by ,*A* and any scalar component ⍰ in ⍴*B* by 1. If *B* is scalar, *Z* is as defined in APL.

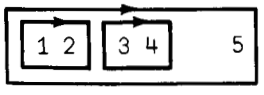
Example:

A



(⍴*A*) ≡ ⁸⁸,2 2
(⍴0 3ρ*A*) ≡ ⁸⁸,0 3_2

B



(⍴*B*)≡⁸⁸,2 ⍰2
(⍴0ρ*B*)≡⁸⁸,0 1 2

Case 2: A has no zero items and *B* is empty.

Z is nonempty and its scalar components are 0. In general, the structure of *Z* is obtained by replacing in the structure of *B* the first item by ,*A*. If *A*≡⍬ and *B* is simple, then *Z* is 0. For example,

If (⍴*B*)≡⁸⁸,0 3
then (2ρ*B*)≡0 0 0_0 0 0
and (⍴2ρ*B*)≡⁸⁸,2 3.

If (⍴*B*)≡⁸⁸,3 0
then (⍴2ρ*B*)≡⁸⁸,2 0.

If (⍴*B*)≡⁸⁸,0 2
then (⍬ρ*B*)≡<0 0
and (⍴⍬ρ*B*)≡⁸⁸,⍬_2.

It is now possible, in order to simplify the exposition, to introduce the function

∇*Z*←*MODEL B* [1]*Z*←⍬ρ0ρ*B*∇

For uniform and nonempty array *B*, the item of *MODEL B* has the structure "typical" of an item of *B* (and the scalar components of *MODEL B* are 0).

Note that *MODEL B* is defined for any array *B*, is always uniform, and if it has scalar components they are 0.

Example: (*MODEL* 1 2 3)≡0.

Actually, for any simple array *A*, (*MODEL A*)≡0. Also

(*MODEL* 1 2_3 4≡<0 0,

(*MODEL* 1 2_3 4__5 6_7 8)≡<0 0_0 0,

(*MODEL* 1 2_3)≡<,0,

(*MODEL* 1 2_3__4_5 6)≡<(,0)⍪,0,

(*MODEL* ⍬)≡0,

(*MODEL* 2 0ρ0)≡0.

If ⍴*B* is ⁸⁸,0 2, then

(*MODEL B*)≡<0 0.

If ⍴*B* is ⁸⁸,0 2 0, then

(⍴ *MODEL B*)≡⍬,⁸⁸,2 0.

It is instructive to compute ⍴*MODEL B* from ⍴*B* as follows. Replace all scalar components ⍰ of ⍴*B* by 1 and replace the first item of ⍴*B*, by ⍬.

The function *reform* is extended to allow its right argument to be an empty array. If *B* is an empty array, then (*A*⍴*B*)≡*A*⍴*MODEL B*.

Example:

(2 3⍴2 0ρ0)≡0 0 0_0 0 0.

The following identity holds, in general:

(⍴*A*⍴*B*) ≡ (⁸⁸,*A*),1↓⍴ *MODEL B*.

Note that the case of *A* having scalar components 0 is defined by the above identity.

The following sections now define the remaining relevant functions.

Catenate

If *A* is an empty vector and *B* is nonempty single/vector,

then $(A,B)\equiv,B$. If both A and B are empty vectors, then $(A,B)\equiv 0\rho(MODEL\ A),MODEL\ B$.
For example,

$(\theta,0\ 2\underline{\rho}0)\equiv 0\ 1\underline{\rho}0.$

Slice
Consider $Z\leftarrow A\downarrow I$. If any of the items of I is a simple empty array, then Z is empty and $(\underline{\rho}Z)\equiv(\cup\ddot{}\rho I)\ \rho\ MODEL\ A$. In case A is empty, at least one of the items of I must be a simple empty array and the above definition holds.

Example:

$(\underline{\rho}1\ 2\_3\ 4\downarrow<\theta)\equiv\ddot{},0\ 2,$

$(\underline{\rho}(2\ 2\rho<1\ 2\ 3)\downarrow 1\ 2\_\theta)\equiv\ddot{},2\ 0\_3,$

$(\underline{\rho}(0\ 2\underline{\rho}0)\downarrow<\theta)\equiv\ddot{},0\ 2.$

If I is θ, then A must be a single and Z is A.

Choose
Consider $Z\leftarrow A\circ I$. If A is nonempty array, then I is an array of indices of A. If A is empty, then I must be empty. In any case, the following must hold:

$(MODEL\ I)\equiv MODEL\ 1\rho A,$
or
$(MODEL\ I)\equiv\ddot{},\ MODEL\ 1\rho A.$

If I is empty, then

$Z\equiv(\rho I)\rho\ MODEL\ A.$

Example:

If $I\equiv 2\ 0\ 3\rho 0,$

$\underline{\rho}(1\ 2\_3\ 4\circ I)\equiv 2\ 0\ 3\_2,$

$(\theta\circ 2\ 0\rho 0)\equiv 2\ 0\rho 0,$

$((0\ 2\underline{\rho}0)\circ 2\ 0\rho 0)\equiv 2\ 0\_2\underline{\rho}0.$

Reach
If I is empty, $(A\underline{\circ}I)\equiv I$. Note that if A is empty, then I must be empty.

Monadic ι
If $0\in I$, then ιI is empty and completely defined by

$(\underline{\rho}\iota I)\equiv\underline{\rho}I\rho<I.$

Dyadic ι
Consider $Z\leftarrow A\iota B$. If A or B or both are empty, then

$Z\equiv(\rho B)\rho<1+\rho A$

except if A is a vector. Then

$Z\equiv(\rho B)\rho 1+\rho A.$

Example:

$(1\ 2\iota 2\ 0\rho 0)\equiv 2\ 0\rho 0,$

$((2\ 2\rho\iota 4)\iota 2\ 0\rho 0)\equiv 2\ 0\_2\underline{\rho}0,$

$(5\iota 2\ 0\rho 0)\equiv 2\ 0\_0\underline{\rho}0.$

Find
Consider $Z\leftarrow A\underline{\iota}B$. In the definition given in the section on generation of indices, if B is not empty, an item K of B that does not occur in A is replaced by $0\rho<\rho A$ (except in the case where A is a vector, when K is replaced by θ). If B is empty, then

$Z\equiv((\rho B)\bar{,}0)\underline{\rho}<\rho A,$

except if A is a vector. Then

$Z\equiv((\rho B)\bar{,}0)\underline{\rho}\rho A.$

Example:

$(1\ 2\iota 2\ 0\rho 0)\equiv 2\ 0\_0\underline{\rho}0,$

$((2\ 2\rho\iota 4)\iota 2\ 0\rho 0)\equiv 2\ 0\_0\_2\underline{\rho}0,$

$(5\iota 2\ 0\rho 0)\equiv 2\ 0\_0\_0\underline{\rho}0.$

Itemwise
For Δ monadic and B empty,

$(\ddot{}\Delta B)\equiv(\rho B)\rho\ddot{}\Delta\ MODEL\ B.$

Example:

$(\ddot{}-\theta)\equiv\theta,$

$(\ddot{}-0\ 2\ 0\underline{\rho}0)\equiv 0\ 2\ 0\underline{\rho}0,$

$(\ddot{}\rho\theta)\equiv\ 0\ 0\underline{\rho}0,$

$(\ddot{}\rho 0\ 2\ 0\underline{\rho}0)\equiv 0\ 2\underline{\rho}0,$

For Δ dyadic and A empty,

$(A\ddot{}\Delta B)\equiv(\rho A)\rho(MODEL\ A)\ddot{}\Delta B,$

and similar relations define the cases where B or both A and B are empty.

Example:

$(\theta\ddot{}\lceil\bar{}1)\equiv\theta,$

$((0\ 3\underline{\rho}0)\ddot{}\rho\theta)\equiv 0\_0\ 0\ 0\underline{\rho}0.$

Reduction
If a function Δ has an identity I then $/\Delta\theta$ is I. Note that catenation and outer product catenation have no identities. We define for an empty B:

$(/,B)\equiv 0\rho>MODEL\ B,$

$(/.,B)\underline{\equiv}MODEL>MODEL\ B.$

Example:

$(/,\Theta)\equiv\Theta$,

$(/.,\Theta)\equiv 0$.

Scan
For an empty vector B,

$(\backslash\Delta B)\equiv 0\rho</\Delta>$ *MODEL B*.

Example:

$(\backslash+\Theta)\equiv\Theta$,

$(\backslash+0\ 2\underline{\rho}0)\equiv\Theta$,

$(\backslash\uparrow 0\ 2\underline{\rho}0)\equiv 0\ 0\underline{\rho}0$.

Outer product
For A empty,

$(A.\Delta B)\equiv((\rho A),\rho B)\rho(MODEL\ A).\Delta B$,

and similar relations define the cases where B or both A and B are empty.

Example:

$(\Theta.\rho 1\ 2\_3\ 4)\equiv 0\ 2\_0\underline{\rho}0$.

Mesh
Recall that $A\divideontimes B$ is obtained by replacing the scalar components of A whose value is K by items of $>(,B)\circ K$. If ravel B has no Kth item or the Kth item of $,B$ is empty, the above replacement is by $>MODEL>MODEL\ B$. If A is empty, then $A\divideontimes B$ is empty and $(\underline{\rho}A\divideontimes B)\equiv(\underline{\rho}A),\underline{\rho}>MODEL>MODEL\ B$.

Example:

$(1\ 3\divideontimes 4\ 5\_6\ 7)\equiv 4\ 0$,

$(1\divideontimes<0\ 2\underline{\rho}0)\equiv<0\ 0$.

Compress
If $0\equiv/+A$, then $(A/B)\equiv 0\rho MODEL\ B$.

Example:

$(0\ 0/1\ 2\_3\ 4)\equiv 0\ 2\underline{\rho}0$,

$(0\ 0/1\_2\ 3\ 4)\equiv 0\ 1\underline{\rho}0$.

Expand
Consider $Z\leftarrow A\backslash B$. If A has a 0 item, then Z is obtained by replacing all 0 items of A by the item of *MODEL B*, and the 1 items of A by the items of B in principal order. If A is empty then Z is B. Precisely, $(A\backslash B)\equiv,(1+\sim A)\divideontimes<B$.

Example:

$(0\backslash\Theta)\equiv 0$,

$(0\backslash 0\ 2\underline{\rho}0)\equiv<0\ 0$,

$(1\ 0\ 1\backslash 1\_2\ 3)\equiv 1\overline{\ }(,0)\overline{\ }2\ 3$,

$(\Theta\backslash B)\equiv B$.

Take
Consider $Z\leftarrow A\uparrow B$. If $/v(|A)>\rho B$ then the items of Z at the additional indices are the items of *MODEL B*.

Example:

$(\overline{\ }3\uparrow 1\ 2\_4\ 5)\equiv 0\ 0\_1\ 2\_4\ 5$,

$(\overline{\ }3\uparrow 1\_2\ 4\ 5(\equiv(,0)\overline{\ }1\_2\ 4\ 5$.

If Z is empty then $Z\equiv(|A)\rho B$.

Example:

$(0\ 3\uparrow 2\ 0\_2\underline{\rho}0)\equiv 0\ 3\_2\underline{\rho}0$.

Drop
Consider $Z\leftarrow A\downarrow B$. If $/v(|A)\geq\rho B$, then

$Z\equiv(0\lceil(\rho B)-|A)\rho B$.

Conclusions

The main purpose of this paper is to introduce as data objects arrays whose items are also arrays. In order to manipulate such arrays, the functions and operators listed in the appendix are defined. The exposition is carried out throughout in terms of APL.

Programming languages such as PL/1 [4], LISP [5], extended GENIE [6], the IBM Vienna PL/1 specification language [7] and SETL [8] allow for complex data structures. In this paper, definitions are given that provide the capability of representing most data structures of the aforesaid languages, while having the simplicity of APL.

However, the discussion in this paper does not exhaust the possibilities opened up by the introduction of general arrays and the general notion of operators. For example, the domains of ϕ and $\circledcirc$ may be extended to include general arrays, and general coordinates in the case of $\circledcirc$. Also, for example, the domain of the axis operator may be extended to include the functions $\not\Delta$, $\forall$ and $\perp$. Further, in this paper only primitive operators (as distinguished from user-defined operators) are discussed, and they are restricted to take arrays or functions as left arguments, functions as right arguments, and to yield functions as results. The removal of these restrictions may be a topic for future research.

A problem of manipulation of files within a data base was mentioned in the introduction. Relatively simple expressions may now be given to answer typical inquiries about files. For example, the vector N of numbers of projects that use part P is obtained by $N\leftarrow((P=B^{\cdot\cdot}\circ 1)/B)^{\cdot\cdot}\circ 2$ and the vector M of names of projects that use part P is obtained by $M\leftarrow A\underline{\circ}((A^{\cdot\cdot}\circ 1)\iota N)^{\cdot\cdot},2$.

Acknowledgments

We are greatly indebted to A. D. Falkoff, K. E. Iverson and T. More, Jr. for many stimulating and instructive discussions during the course of this work.

References

1. Trenchard More, Jr., "Notes on the Development of a Theory of Arrays." See the bibliography in IBM Philadelphia Scientific Center Report No. 320-3016, March 1973.
2. T. More Jr. "Axioms and Theorems for a Theory of Arrays". *IBM J. Res. Develop.* 17, 135 (1973).
3. A. Falkoff and K. Iverson, *APL/360 User's Manual.* IBM Corporation G-h20-0683, 1968.
4. *IBM System/360 PL/1 (F) Language Reference Manual.* IBM Corporation (GC28-8201-3) 1970.
5. J. McCarthy, LISP *1.5 Programmer's Manual.* Computation Center and Research Laboratory of Electronics MIT, Cambridge, Mass., August 1962.
6. G. A. Sitton, "Operations on Generalized Arrays with the Genie Compiler," *CACM* **13,** 284-286 (May 1970).
7. P. Lucas, P. Lauer, and H. Stigleitner "Method and Notation for the Formal Definition of Programming Languages," IBM Laboratory Vienna, Technical Report TR25.087, June 1968. Revised July 1970.
8. J. Schwartz, "Set Theory as a Language for Program Specification and Programming," Notes.

General reference:

J. Brown, "A Generalization of APL," Ph. D. Dissertation, System and Information Science, Syracuse University, September 1971.

Received May 25, 1972

The authors are located at the IBM Data Processing Division Scientific Center, 3401 Market Street, Philadelphia, Pennsylvania 19104.

Appendix. Functions and operators

	Symbol	*Monadic*	*Dyadic*
FUNCTIONS	<	enclose *	
	>	disclose *	
	,	ravel	catenate
	⍪		link *
	⍪		pair *
	⍴	shape	reshape
	⌽		slice *
	○		choose *
	⍜		reach *
	⍳	iota	iota
	⍳		find *
	⍴	form *	reform *
	↑		raise *
	↓	collapse *	lower *
	∪	unravel *	
	⍮		level *
	⌸		mesh *
	/		compress
	\		expand
	↑		take
	↓		drop
	≡		identical to *
	∈		membership
	∈		item of *
	~		less *
	⊗	combine *	
OPERATORS	¨	itemwise *	
	/	reduction	
	\	scan	
	.	outer product	inner product
	⌸	limit *	fold *
	:		axis *

*These functions or operators are new.