From Trees into Boxes

David Steinbrook
725-L Blair Court
Sunnyvale, CA 94087

Eugene McDonnell
1509 Portola Avenue
Palo Alto, CA 94306

Abstract

This paper is a progress report on work undertaken to include tree data structures by means of the boxed data type available in J. Methods for displaying these boxed arrays as trees are shown. This work is part of a larger effort to provide a comprehensive set of facilities in J for working with tree structures. The facilities described were at first modelled in J and subsequently translated into C, in order to provide a J interpreter which has trees as native facilities. Thus this work also exemplifies the way in which one can tailor the J interpreter to special needs.

Introduction

Tree structures have been used for centuries to represent such things as families and hierarchies. They entered science with the work of Kirchoff and Von Staudt in the 1840s, and became the object of mathematical study themselves with the work of Cayley in the latter half of the century, who used them in the analysis of algebraical formulas. Iverson [Iv62] discusses trees as specializations of graphs; that is, they are graphs containing no circuits and having at most one branch entering each node. Their use in computer algorithms gives them great current importance. Knuth [Kn68] calls trees "the most important nonlinear structures arising in computer algorithms."

There have been numerous attempts to provide trees in the programming language APL in the past. The comprehensive survey article by Ruehr gives pointers to many of these [Ru82].

The work presented here exploits the well-known relationship between lists of lists and trees [Br71,Ry71, Ed73, Mu73]. Rather than introduce a new data type, it uses the existing boxed array of J to obtain its tree representations. It differs from the similarly motivated work of Nauta [Na87] in several respects, most fundamentally in that a tree becomes a single boxed object rather than a list of boxed lists wherein successive boxed items in the list are used to hold the nodes at successive levels.

This work had its origin in a desire to include trees in APL systems. Although Iverson's early book [Iv62] dwells at great length on trees as data structures, APL implementations have not provided them as a data type. One reason for the omission was that tree structures in all the referenced works are *flat* trees: the analogy in nature is the highly artificial *espalier*, or tree forced against a wall. Iverson envisions trees as *round* at each node; the dispersion to other nodes may occur in any direction, which is to say, with any rank and shape [Iv92]. Such rounded trees may be viewed as the general case, which leaves the familiar flat trees as a subset.

In any tree, round or flat, the basic atom is called a node, which has a value, and zero or more children. A node that has no children is called a leaf. We can easily model flat trees and operations upon them, then extend the model to round trees as a proper representation for their generation and display emerges.

For more than a decade, this work on trees proceeded on the assumption that the way to model flat trees was to isolate them from the rectangular arrays in APL and treat them as a separate data type with separate facilities to manipulate them. With the advent of J, for which C source code with very important high level properties is available as freeware, it became possible to go the next step, from APL models to actual interpreter code, and, in fact, an interpreter was built in which the tree data type had been added.

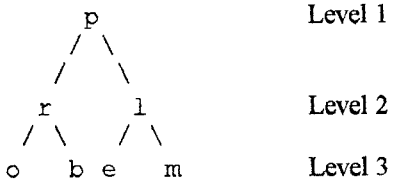
However, subsequent discussions with Iverson and Hui [Iv92] about this interpreter led to the decision to set this work (which introduced the tree as a new data type) aside and replace it with an interpreter in which trees were obtained as a subset of J's existing boxed data type. This was in part in recognition of the large overlap in

functionality between the new tree primitives and existing rectangular primitives; it was also in recognition of the fact that rounded trees are in fact extended rectangular structures, and traditional structural verbs (such as shape and reshape) would eventually have to extend to them.

A J interpreter has now been built in which trees and the primitive verbs, adverbs, and conjunctions which manipulate them are made available without adding a new data type to the language.

Flat Tree Representation

We can show a flat tree pictorially:



This tree consists of the values `'p'`, `'r'`, `'l'`, `'o'`, `'b'`, `'e'`, and `'m'`, and a structure holding them in a certain arrangement. When we describe the structure, we find each value at a certain level, as indicated. Level 1 is the root level. We can move along the branches of the tree in a variety of ways. We shall use the *forward walk* or *preorder* convention [Kn68], in which we begin at the root and go along the left branch until we reach a leaf, recording the level of each node we meet as we are moving down. When we reach a leaf we go back to the preceding node and take the next branch, if there is one; if there is none, we go back to the next higher node, and so on. For the tree shown, we proceed as follows:

Start at value `p`	Append 1	1
Go down to value `r`	Append 2	1 2
Go down to value `o`	Append 3	1 2 3
Go up to value `r`		
Go down to value `b`	Append 3	1 2 3 3
Go up to value `r`		
Go up to value `p`		
Go down to value `l`	Append 2	1 2 3 3 2
Go down to value `e`	Append 3	1 2 3 3 2 3
Go up to value `l`		
Go down to value `m`	Append 3	1 2 3 3 2 3 3
Go up to value `l`		
Go up to value `p`		
End		

The list we end with is called a *depth list*. Iverson referred to it as a *precedence vector* [Iv62]. It gives the depth of each node we meet as we go down the tree along the leftmost branch we haven't yet taken. It can be shown that the depth list characterizes the structure of the tree uniquely. Each item in the depth list gives the depth of the corresponding node in the order of the walk.

A *connection table* is a directed graph representation of the same structure, and is easily derived from the depth list. The verb `CD` produces a connection table from a depth list.

```
   CD d=.1 2 3 3 2 3 3
0 1 0 0 1 0 0
0 0 1 1 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 1 1
0 0 0 0 0 0 0
0 0 0 0 0 0 0
```

A third representation of tree structure used extensively by Iverson provides a basis for tree indexing. The representation is the *left list*, familiar from scientific papers as the numeric labels (most often using origin 1) attached to outlines. The verb `LD` produces a left list (in zero origin) from a depth list.

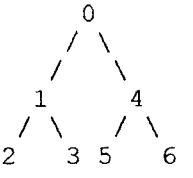
```
   ]c=. LD d
```

0	0 0	0 0 0	0 0 1	0 1	0 1 0	0 1 1

The open of the formatted entries in a left list reveals the origin of the term:

```
   > ":&.> c
0
0 0
0 0 0
0 0 1
0 1
0 1 0
0 1 1
```

The same tree structure can be shown with the values in each node replaced by the order in which each first appears in a forward walk:



A simple explanation of the construction which is the basis of this paper can be given in terms of a depth list `d` of integers and a value list `v` of boxed items. The depth list is already defined.

```
   d
1 2 3 3 2 3 3
```

For the value list we'll box the letters of the word

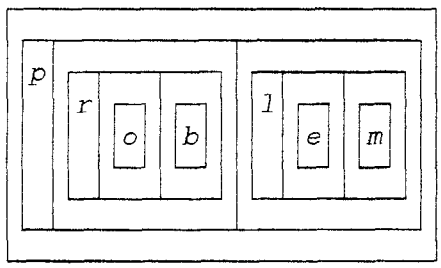
problem, using the verb bs, which boxes the atoms of its argument:

```
   bs =. <"0
   ]v=. bs 'problem'
```

p	r	o	b	l	e	m

Imagine a dyadic verb entree which takes as left argument a depth list d and as right argument a value list v, producing a boxed array T corresponding to the tree shown above:

```
   ]T =. d entree v
```



T is an atom with contents a list of three boxes: the value at the root, and its two child nodes. The two child nodes are subtrees each of which has two child nodes. These are leaf nodes (have no children).

Tree Display

There are several different ways to display a tree. The two ways provided here are the outline form and the chart form. The verb outline, given a noun such as T, displays it in the form of a table of contents, in which the amount of indentation from the left margin corresponds to the level of the node (as in section, sub-section, sub-sub-section, and so forth):

```
   outline T
p
   r
      o
      b
   l
      e
      m
```

The verb outline may be controlled in detail through a left argument which affects the amount of indentation, and the presence and type of left list labels affixed to the display.

The verb chart, applied to T, displays it in the form of an organization chart in which the root node is shown at the top, and successively lower-level nodes in successively lower regions of the display:

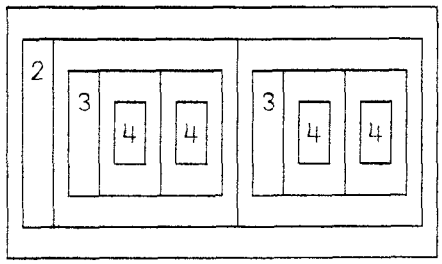
```
   chart T
      p
  r       l
o   b   e   m
```

The verb chart also has a dyadic case which allows stages in the generation of final output to be captured in the result.

Form of Arrays

The monad *form* [Hu89], applied to any boxed array, returns an array of the same depth structure in which the level of depth of each item appears instead of the value of the item. We call this the *form* of the array, where form is the depth analog of shape.

```
   form T
```



Applying the verb flatten [Hu89] to any boxed array removes the depth structure and presents the items at the first level of depth. When applied to a tree, the verb flatten gives a list of boxes containing the values in the order of items in the depth list:

```
   flatten T
```

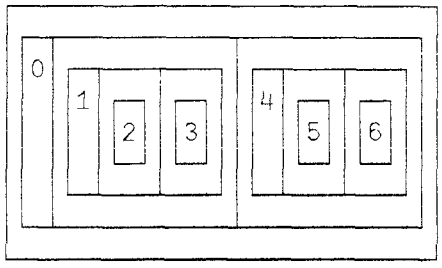
p	r	o	b	l	e	m

The verb flatten, applied to the form of a tree, yields the boxed level numbers as a list. Opening this list (with >) and decrementing the result (with <:) yields the depth list:

```
   <: >flatten form T
1 2 3 3 2 3 3
```

Just as *shape* has a dyadic case called *reshape*, so the dyadic case of *form* is called *reform*. The verb reform [Hu89], with left argument a form and right argument a list of boxed values, yields an array with form corresponding to the left argument, and values taken from the right argument.

```
(form T) reform <"0 i.7
```



The Implementation

Source code for the J interpreter is available free. This makes it an attractive vehicle for experiments of the kind we envisioned. The J source code is written with the aid of a set of C preprocessor definitions and macro instructions which augment the expressive power of C, enforce uniformity, and increase readability [Hu90]. It also makes it possible for a non-systems programmer to undertake serious system programming tasks, which is what we have done.

There is a facility in J analogous to the I-beam of APL\360, but with a difference. The foreign conjunction (`!:`) provides an open-ended set of facilities, intended primarily for communicating with the host system as well as with the keyboard and with the screen. The arguments are typically nonnegative integers, and facilities associated with a particular set of tasks have the same left argument. Thus, the left argument for session operations is 0, for file operations is `1`, for workspaces is `2`, for names is `4`, and so forth. The right arguments are similarly associated, where possible, with similar operations. For example, the verb to end a session is `0!:55`; to erase a file is `1!:55`; to erase an object from a saved workspace is `2!:55`; and to erase an object from the current workspace is `4!:55`.

This foreign conjunction may also be used, as in our case, to provide special facilities unique to our needs. It is straightforward to add to a table in the interpreter an entry which defines new left and right arguments for this conjunction. We have taken the left argument `12` to define a family of tree verbs, so that `12!:n`, for some integer `n`, signifies one of the tree verbs.

We have defined a great many special tree primitives, in order to facilitate experimentation. These are provided to the user, not only as cases of the `12!:n` verb, but also with a suite of *given names*, that is, names ending with a colon, one for each case of `12!:n`. Such names can be assigned only once; thereafter an attempt to reassign the name will be rejected (unless the name has been erased since being assigned).

As is usual in this kind of design, the process begins with the making of a J model for the verb we desire. After we are convinced that the model performs as we wish the primitive to do, we translate the J model into the high level C available through the J interpreter.

To add tree verbs to the J interpreter is straightforward. For example, the first entry in our C file reads:

```
case XC(12,0) : R CDERIV(CIBEAM,
   Form,Reform, RMAXL, RMAXL, RMAXL);
```

This says that we are adding a new `!:` case, using the C primitive `case` within a `switch` statement. `XC` encodes the left and right arguments 12 and 0. `R` is shorthand for the C `RETURN` primitive. `CDERIV` derives a verb from a conjunction; the conjunction being `CIBEAM` (that is, `!:`). The monad and dyad cases of the derived verb are `Form` and `Reform`, respectively; the rank of the monad is maximum (`RMAXL`), as are the left and right ranks of the dyad. A large amount of system programmer apparatus is secreted in `XC`, `CDERIV`, and `CIBEAM`, that we don't have to worry about.

The J model defines `form` as a special case of the verb `fo1`, having fixed left argument 0. Its text is:

```
($y.)$ 0 fo1 y.
```

The verb `$` stands for the monad *shape* and the dyad *reshape*. The verb `fo1` has a simple definition which we need not discuss here.

The C version adheres closely to the J version. The convention used is that in a monad the name `w` refers to the argument. A verb definition typically has a few lines of preliminary code, having to do with allocating storage for the C function, providing temporary name definition, value and type tests, and ends with some cleanup code to deallocate the temporary storage. In the case of the *`form`* verb, the line of C code corresponding to the J text is as follows:

```
z=reshape(shape(w),fo1(zero,w));
```

Aside from converting symbols to names, and adapting to C's prefix notation, this is identical to the J code. The macros provided for the J interpreter make writing in C very close to writing in J.

Facilities for Tree Manipulation

The following list gives the name of each facility we have provided, and a short description of its purpose. We intend to provide a further paper to describe these in more detail.

`Amend:`	Replace subtree at given location with another
`Behead:`	Drop the first item
`Cat:`	Catenate argument trees as subtrees at the root
`Catalog:`	Generate left list indices for argument path
`CD:`	Collection table from depth list
`Chart:`	Display as organization chart (levels top down)
`Conceal:`	Drop levels from root (+) or leaves (-)
`Curtail:`	Drop the last item
`DC:`	Depth list from Connection table
`DF:`	Depth list from Form
`DL:`	Depth list from Left list
`Disperse:`	Expression from token tree (see `Gather:`)
`Drop:`	Drop indicated items
`FD:`	Form from depth list
`Flatten:`	Remove boxing structure (depth ravel)
`Form:`	Return boxing structure (depth shape)
`From:`	Select nodes, leaves, paths, and other extracts
`Gather:`	Token tree from expression (see `Disperse:`)
`GCD:`	Intersection of argument depth lists (see `LCM:`)
`Graft:`	Introduce subtree at node
`Head:`	Return the first item
`IC:`	Item count (number of subtrees at the root)
`IndOf:`	Index of, dyadic iota for tree arguments
`Invert:`	Transpose: tree to forest or forest to tree
`Item:`	Adverb to apply a verb to items (subtrees)
`LCM:`	Union of argument depth lists (see `GCD:`)
`LCat:`	Conjunction catenates trees at given level
`LD:`	Left list from depth list
`Lamin:`	Catenate tree arrays below new (empty) root
`LeafScan:`	Conjunction; verb consolidates from leaves
`Level:`	Select by level
`Locate:`	Adverb returns node indices for selection verb
`Match:`	Two trees match in form and content
`Member:`	If node or item is contained in argument
`NodeFrom:`	Select subtree from given node to leaves
`NodeTF:`	Select subtree to and from given node
`NodeTo:`	Select subtree to and including given node
`Normal:`	Normalize depth list
`Nub:`	Return unique items (subtrees at the root)
`NubS:`	Return Boolean list to select unique items
`Outline:`	Display tree as outline (levels left to right)
`OverTake:`	Tree with empty nodes based on depth supplied
`PathFrom:`	Select subtree with given path
`Pervade:`	Adverb to apply any verb to boxed argument(s)
`Prune:`	Remove a subtree from a tree
`Reform:`	Impose new boxing structure (depth reshape)
`Repeat:`	Repeat items (subtrees at the root)
`Reshape:`	Recycle items (subtrees at the root)
`Reverse:`	Reverse items (subtrees at the root)
`RootScan:`	Adverb applies verb along paths from the root
`Rotate:`	Rotate items (subtrees at the root)
`Tail:`	Return last item
`Take:`	Return indicated items (head or tail) or subtree
`Tree:`	Test depth list for tree connectivity

J Listings

The verbs presented in this paper are here listed, and in certain cases, annotated.

Form Verbs

The form verbs are `form`, `flatten`, and `reform`. All were originally written in the SAX system (Sharp APL for UNIX) using direct definition. In J, the agenda conjunction (`@.`), with its embedded case statement, is a good mechanism for these concise definitions. The test verb in the right argument of agenda takes the place of the middle expression in the direct definition. A `0` or `1` result of this test selects the first or the second verb in the gerund left argument. The general expression for this binary switch, `v1`v2 @. test`, appears in the subverbs of `form` and `reform`, in `flatten` itself, and in various other verbs to follow.

Form

```
form=.$@] $ 0&f1@]
f1=.[`f2@.(boxed@])
boxed=.32&=@(3!:0)
f2=.f1"0`(>:@[ f1&.> ])@.(ft2@])
ft2=.0: = #@$
```

Flatten

```
flatten=.<`(;@(flatten&.>@,))@.boxed
```

Reform

```
reform=.<"0@[r1&>$@[$,@(([rrf])<;.1])
rrf=.i.@(#@])e.+/\@(0:,,@(count"0@[))
r1=.r1g1`r2@.(boxed@[)
r1g1=.>@({.@])
boxed=.32&=@(3!:0)
r2=.reform`r2g2@.r2t
r2g2=.<@(>@[ r1 ])
r2t=.0:=(#@($@[))
count=.1:`(+/@count&.>)@.boxed
```

Depth Conversion Verbs

The depth conversion verbs treat the depth list as reference representation for tree structure. They convert a depth list to the three alternate forms: left list, connection table, and form; their inverses are also supplied. We adopt a 2-letter naming convention: the first letter is a descriptor for the result, and the second a descriptor for the argument. The letter `D`, which appears in all names (either as argument or as result), stands for depth list; `C` stands for connection table; `L` stands for left list; and `F` stands for form.

Connection Table from Depth List

The verb `CD` produces a connection table from a depth list.

```
CD=.</\&.|.@(</~@(i.@$) * </~)
```

Depth List from Connection Table

The inverse verb, DC, transforms a connection table to a depth list. It is based on the pairwise difference fork }.-}: and appears as three verbs for clarity of presentation:

```
p=.}.-}:
DCf=.0:,}.@>:@(i."1&1@|:)
DC=.>:@(+/\@(1:<.0:,p@DCf))
```

Depth List from Left List

The verb DL converts a left list to a depth list:

```
DL=.#&>
```

Left List from Depth List

The inverse verb, LD, takes as argument a depth list and returns a left list (in zero origin). The depth list is a list of numbers, and the left list is a list of boxes (because the items contain lists of varying lengths). For example, for a depth list 1 2 3 3 2 3 3, the result is

0	0 0	0 0 0	0 0 1	0 1	0 1 0	0 1 1

We assume the argument list has at least one element, that the leading item is 1, that all items are positive, and that an item is no more than 1 greater than its predecessor. A depth validation verb (Tree:) that tests for precisely these constraints is a part of the proposed set of tree facilities.

Let us construct an explicit definition for the verb, which we call ELD. The initial result, true for all valid arguments, is a boxed one-element list 0

```
z=.<,0
```

We fall through if the result is shorter than the argument (i.e. more result is needed) otherwise go to the end (the result is finished):

```
L0)$.=.>((#z)=#y.){$.;LZ
```

Obtain the current and previous elements of the argument

```
e=.(_1 0+#z){y.
```

For the initial value of the next element of the result, use the current element of the argument to *take* from the *open* of the previous result. For example, if the previous result element is 0 1 2 and the new argument element is 1, yield 0; if the argument is 2, yield 0 1; if 3, yield 0 1 2; and if 4, yield 0 1 2 0.

```
r=.({: e){.>{: z
```

Add one to the last element of this if the current value of the argument is less than or equal to the previous value. For example, if the previous and current values are 3 1, yield 1; if 3 2, yield 0 2; if 3 3, yield 0 1 3; and if 3 4, yield 0 1 2 0.

```
c=.((>:/e)+{:r)_1} r
```

Box this and catenate it to the result

```
z=.z,<c
```

Go back to compute the next element of the result

```
$.=.L0
```

Yield the result

```
LZ)z
```

The entire verb ELD is listed below for convenience. It follows a definition originally proposed by R. H. Lathwell in 1979.

```
   z=.<,0
L0)$.=.>((#z)=#y.){$.;LZ
   e=.(_1 0+#z){y.
   r=.({: e){.>{: z
   c=.((>:/e)+{:r)_1} r
   z=.z,<c
   $.=.L0
LZ)z
```

The explicit verb ELD is very close in spirit to what one would write in APL. There is a label at the top of a loop with a conditional branch (taken in the case of termination, and fallen through in the case of continuation), and an unconditional branch at the bottom of the loop. Thus in J one can write programs like conventional APL programs. In J, however, there is another way to write programs, akin to the method called *functional programming*, and called in J *tacit definition*. In this method, one writes a program (*verb*) entirely without explicit reference to the arguments, and without the use of local variables, using functionals (verbs, adverbs and conjunctions) only. This method is worth learning to exploit, and we now describe a version of the verb to convert a depth list to a left list, written in *tacit* form. Because a tacit verb requires no parsing, LD is more than three times faster than the explicit verb ELD.

We think of LD as comprising three stages: argument preparation, for which we provide the verb pa; the processing stage, for which we use a verb ps; and result post-processing, for which we use a verb pr. These are used one after the other, and we can connect the verbs with the *composition* conjunction, denoted by @ . Thus we can write

```
LD=.pr @ ps @ pa
```

Within ps, we proceed by steps. At each step, we use take based on the absolute value of the depth item. We add one to the last element produced by take if the current depth item is less than or equal to the previous one. If we think of the result being obtained by repeated applications of a single

dyadic verb k, we can describe the process with this pattern:

d_{n-1} k ... k d_1 k d_0 k dr

where dr is a catalyst representing the initial state of the (boxed) result, but not entering into the final result; d_i is the value of the i-th item of the argument; and k is the verb which does the processing. The items of d are the items of the argument in reverse order, so we reverse (|.) the argument. The result of d_0 k dr becomes the right argument to the next use of k, and so on. This pattern suggests that, since the verb k is inserted between each noun item, we could use the insert adverb (/) with k to accomplish the processing. We rewrite our processing pattern:

```
k / (|.d) , dr
```

We can't really write this, however, because J permits only homogeneous arrays, and d and dr are of different types. To get around this, we box the items of d so that we can directly append dr to the now-boxed items. The boxed array also means that we can't use k directly, but must use the verb k in composition with open k&.> (where &. is dual, and > is open) as the verb to be used with the insert adverb

```
ps =. k&.> /
```

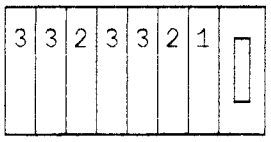
We can now define the verb pa.

```
pa =. ,&(<<'')@(<"0@|.)
```

To review, the argument is reversed (|.), its items ("0) are boxed (<), and a doubly boxed empty list (<<'') is appended (,). Why doubly boxed? A first boxing is needed to account for the use of dual to allow the depth list to be joined to the initial result. A second boxing is needed because the items of the result must themselves be boxed. The value of the doubly boxed item must have tally less than the leading item of the depth list; since this is 1, the tally must be 0. A suitable catalyst, then, is <<''.

Applying pa to a depth list argument gives us the prepared argument, ready for processing:

```
pa d
```



The central processing step ps uses as left argument an element of the depth list, and as right argument the result list accumulated so far. Open the last item of the right argument (>@{:@]). Use the verb take with this value as the left argument. This take produces the fledgling result item, the augend, of the proper length, and with all but its last item determined. For example,

```
   2 {. 0 2 1
0 2
   3 {. 0 2 1
0 2 1
   4 {. 0 2 1
0 2 1 0
```

Its final element has 1 added to it if the current depth list item is less than or equal to the length of the last result item, after being opened.

```
   2 (<:#) 0 2 1
1
   3 (<:#) 0 2 1
1
   4 (<:#) 0 2 1
0
```

The addend can be formed by performing a negative take with the left argument on this Boolean value:

```
   _2 {. 1
0 1
   _3 {. 1
0 0 1
   _4 {. 0
0 0 0 0
```

The augend and addend can be added, yielding the new result item. The entire process can be seen as follows:

```
   2({.+-@[{.#@]>:[)0 2 1
0 3
   3({.+-@[{.#@]>:[)0 2 1
0 2 2
   4({.+-@[{.#@]>:[)0 2 1
0 2 1 0
```

The verb g used dyadically produces the desired result:

```
g =. {. + -@[ {. (<:#)
```

The dyadic hook (<:#) produces the result as shown in the column labelled *Test* below. The result of take ({.) with a negative left argument (-@[) is shown in the *Addend* column. Combining the augend and the addend with add (+) produces the *Sum* column:

Left Arg	*Augend*	*Test*	*Addend*	*Sum*
1	0	1	1	1
2	0 2	1	0 1	0 3
3	0 2 1	1	0 0 1	0 2 2
4	0 2 1 0	0	0 0 0 0	0 2 1 0

A verb h uses g and has as its purpose opening (>) the last item ({:) of the right argument (]):

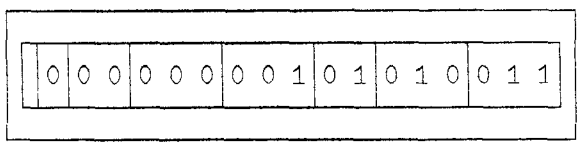
```
   h =. [ g >@{:@]
   1 h <0 2 1
1
   2 h <0 2 1
0 3
   3 h <0 2 1
0 2 2
   4 h <0 2 1
0 2 1 0
```

Now k can be written, using h:

```
   k =. ] , <@h
```

The new verb boxes (<) the result of h and appends it to the right argument (]). We can use k to obtain a preliminary to the final result:

```
   (ps @ pa) d
```

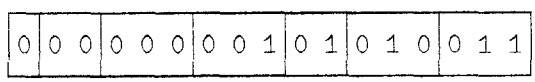


The result shown above has an extra level of boxing which must be removed. The leading element is the catalyst, which must also be removed. These conditions determine the post-processing verb pr, which needs to open this array (>), then behead it ({.).

```
   pr =. }.@>
```

The verb LD can now be written:

```
   LD =. pr @ ps @ pa
   LD d
```



The constituent verbs of LD are listed here for reference:

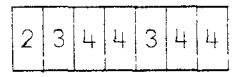
```
   LD =. pr @ ps @ pa
   pr =. }.@>
   ps =. k&.> /
   k =. ] , <@h
   h =. [ g >@{:@]
   g =. {. + -@[ {. (<:#)
   pa=.,&(<<'')@(<"0@|.)
```

Form from Depth List

The verb FD generates a form from a depth list. It can be derived very simply from entree (described below). Add one to the items in a depth list using the increment verb (>:), yielding

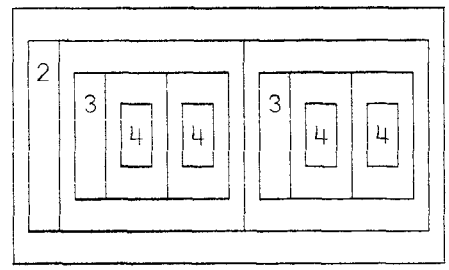
```
2 3 4 4 3 4 4
```

and box (<) the atoms ("0), yielding



When this boxed list becomes the right argument to entree, with the original depth list as the left argument, the result is the form of the tree array with the given depth list. This can all be achieved by defining FD as the hook

```
   FD=.(entree <"0)@>:
   FD 1 2 3 3 2 3 3
```



Depth List from Form

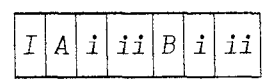
The verb DF produces a depth list from a form.

```
   DF=.<:@>@flatten@form
```

Entree

The verb entree takes as left argument a depth list and as right argument a conforming list of boxes. It returns a tree with structure corresponding to the left argument and values corresponding the right argument. For example, let the right argument be

```
   ]y=.'I';'A';'i';'ii';'B';'i';'ii'
```

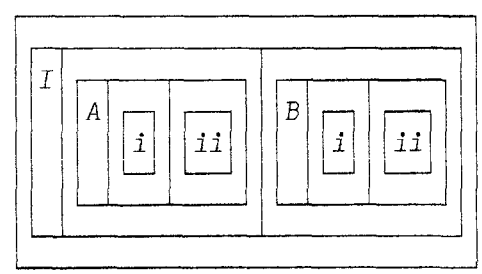


and the left argument the depth list

```
   x=.1 2 3 3 2 3 3
```

Then

```
   x entree y
```



The definition of entree [Hu93] is

```
   entree=.et @ (<"0@[,.])
```

which is a composition (@) in which a verb et is applied to

a verb which boxes (<) the items ("0) of the left argument ([) and appends (,.) the result to the right argument (]). For example, if the left argument is the depth list x, the result of applying <"0 to x is

1	2	3	3	2	3	3

When this boxed list is appended to the boxed right argument shown above, the result is

1	*I*
2	*A*
3	*i*
3	*ii*
2	*B*
3	*i*
3	*ii*

which is the argument that et sees. The verb et

```
et=.et0`et1 @. ett
```

is an *agenda* in which the gerund contains two cases (et0 and et1). The test verb ett determines which case is to be executed, and is defined as

```
ett=.*@#
```

The verb ett applies the signum (*) atop (@) the tally (#), and yields 1 if there are 1 or more items in the argument, and 0 if there are no items. When there are no items in the argument, the first verb in the gerund is executed. When there are 1 or more items in the argument, the second verb in the gerund is executed.

The first verb, called et0, applied to any argument yields the empty list.

```
et0=.i.@0:
```

The second verb, called et1, is used recursively within the verb et.

```
et1=.mask <@({:@{. , et@}.);.1 ]
```

In the definition of et1, we find a fork in which the left verb is mask (described below) which yields a logical list, and the right verb is the identity verb (]), which returns its argument unaltered. The central verb is an instance of the cut-1 adverb (;.1), applied to the verb

```
{:@{. , et@}.
```

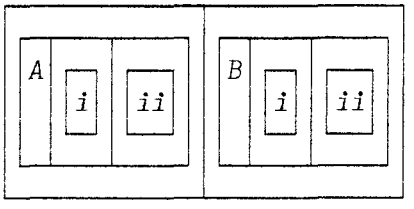
This verb is also a fork in which the left tine selects the tail ({:) of the head ({.) of the argument, and appends this to the recursive use of et applied to the behead (}.) of the argument. For example, if the argument is the two column table shown above, the tail of its head is

I

and the behead of the argument is

2	*A*
3	*i*
3	*ii*
2	*B*
3	*i*
3	*ii*

The verb et applied to this yields the two element list



which is appended to the result of the left verb to produce a result which itself is boxed, yielding the final result shown above.

In the definition of the verb mask,

```
mask=.(={.)@:({."1)
```

the head ({.) rank-1 ("1) verb selects, as atoms, the leading item of each row of its table argument, yielding

1	2	3	3	2	3	3

To this is applied the hook (={.), or equals (=) head ({.), so that the head

1

is compared for equality with the whole, yielding the logical

list

```
1 0 0 0 0 0 0
```

which the left argument to the verb derived from the `cut-1` adverb. On the next pass through the algorithm, the argument is

2	A
3	i
3	ii
2	B
3	i
3	ii

The `mask` verb operates on the first column,

2	3	3	2	3	3

which is a forest (with two subtrees) and produces the logical list

```
1 0 0 1 0 0
```

When this logical list is used with the verb derived from cut, it applies to each of the subtrees, and so on, recursively, until all nodes are processed.

The set of verbs used in defining `entree` is:

```
entree =. et@(<"0@[ ,. ])
et=.et0`et1@.ett
et0=.i.@0:
et1=.mask <@({:@{. , et@}.);.1 ]
ett=.*@#
mask=.(={.)@:({."1)
```

References

[Br71] Brown, James A., A Generalization of APL, Ph. D. thesis, Department of Systems and Information Science, Syracuse University, 1971

[Ed73] Edwards, E. M., Generalized arrays (lists) in APL, Proceedings APL Congress 73, Copenhagen 1973, pp 99-105

[Hu89] Hui, R. K. W., SAX Models for Form, Reform, and Flatten. Private communication.

[Hu90] Hui, R. K. W., K. E. Iverson, E. E. McDonnell and Arthur T. Whitney, APL\?, APL90 Conference Proceedings, ACM, Copenhagen 1990

[Hu93] Hui, R. K. W., J Model of Entree, private communication

[Iv62] Iverson, K. E., A Programming Language, Wiley, 1962, Section 1.23 Ordered Trees, pp 45-62, and Section 3.4, Representation of Trees, pp 121-128

[Iv92] Iverson, K. E., Private communication, 1992

[Kn68] Knuth, Donald E., Fundamental Algorithms, Section 2.3, Trees, Addison Wesley, 1968, pp 305-406

[Mu73] Murray, Ronald C., On tree structured extensions to the APL language, Proceedings APL Congress 73, Copenhagen 1973, pp 333-338

[Na87] Nauta, G. C., Trees as nested arrays and the use of under-disclose, APL87 Conference Proceedings, Dallas 1987, pp 157-162

[Ru82] Ruehr, Karl Fritz, A Survey of Extensions to APL, APL82 Conference Proceedings, Heidelberg 1982, pp 277-314

[Ry71] Ryan, Jim, Generalized Lists and Other Extensions, APL Quote-Quad 2, No. 1, 1971, pp 8-10

Colophon

This paper was prepared in Microsoft Word for Windows as a client application in communication with J (v6.2) as a Windows server through dynamic data exchange (DDE).