Nested Arrays

Summary

1. Introduction

2. Creating Test Data

The primitive functions and operators illustrated in this section are:
Enclose: Monadic <
Each: Monadic operator denoted by ¨ that applies to monadic and dyadic functions

In addition, some syntax specific to the formation of nested arrays appears, called strand notation.

3. Characteristics of Nested Arrays

The primitive functions illustrated in this section are:

Shape: Monadic ρ
Indexing: [;⍳;]
Count and Choose: Monadic and Dyadic #
Type: Monadic ∨
Depth: Monadic ≡
Apply: Monadic operator denoted by ¨ that applies to function scalars

3a. Nested Arrays, Function Arrays and Symbol Arrays

3a1. Type

3a2. Depth

3b. General Purpose Primitive Functions and Nested Arrays

4. Constructing Nested Elements and Extracting their Contents

4a. Slot-Filler Arrays

5. Primitive Functions and Operators Specifically for Nested Arrays

5a. Partition, Partition Count, and the Each Operator

5b. Raze and Rake

1. Introduction

In the Simple Arrays tutorial we concentrated on homogeneous arrays whose elements were integers, floating-point numbers, characters, or symbols. The emphasis was on the items of an array, and how the concept of items, and more generally those of frames and cells, provide a unifying view of simple arrays for understanding the behavior of many primitive functions. In this chapter - which builds on the Simple Arrays tutorial - we turn to nested, heterogenous arrays, i.e. arrays whose elements are other arrays or even function expressions.The primitive functions discussed in the Simple Arrays tutorial that select or rearrange their arguments apply to nested arrays as well as simple arrays; examples will be given. In addition, their are primitive functions, and one primitive operator, that are most meaningful when applied to nested arrays. The more complex data structures that are possible with nested arrays provide alternative ways to approach many programming problems.

Experimentation is still the best way to learn, and so we begin again with methods of creating test data.

2. Creating Test Arrays

The primitive functions and operators illustrated in this section are:
Enclose: Monadic <
Each: Monadic operator denoted by ¨ that applies to monadic and dyadic functions

In addition, some syntax specific to the formation of nested arrays appears, called strand notation.

As with simple arrays, it is helpful to know how to produce test data for heterogeneous, nested arrays.

Examples

Execute each of the following:

```
<⍳3 5 ⍝ A scalar holding the integer matrix ⍳3 5,
⍝   or what is called the enclose of ⍳3 5⌈.
```

The Enclose primitive applies to any array and produces a nested
scalar holding that array.

```
(10 32;'abc') ⍝ Strand notation for a 2-element vector whose
⍝ elements hold the vectors 10 32 and 'abc'.
```

Note the symbol < to the left of both 10 32 and 'abc' in the display
of the result of this expression. This symbol indicates that the
array to the right of the symbol is enclosed within some other array.

```
((10 32;'abc');2 2⍴'ABCD') ⍝ Strand notation for a 2-element
⍝ vector whose first element holds
⍝ the nested vector (10 32;'abc'),
⍝ and whose second element holds
⍝ 2 2⍴'ABCD'.

2 2⍴(10.5 113.58;'abc';`xft3;⍳2 4)  ⍝ A 2x2 matrix consisting of a
⍝ floating-point vector, a
⍝ character vector, a symbol
⍝ scalar, and an integer matrix.
```

All the vectors in these examples are created by strand notation. In
general, a strand is simply a list of expressions separated by
semicolons, with the entire list surrounded by parentheses.

Note that the display of the result of the last expression does not
reflect its shape; the elements are simply listed one above the other
in row major order. A more effective display is produced the function
disp.lay in the script disp.a. Execute the following to see these
displays.

```
$load disp
disp.lay (10 32;'abc')
disp.lay 2 2⍴(10.5 113.58;'abc';`xft3;⍳2 4)
```

The tutorial will not explicitly refer to this display function again,
but you should feel free to use it to clarify results.

```
(?10⍴50;?15⍴32) ⍝ A 2-element vector consisting of
⍝ a vector of 10 random numbers
⍝ between 0 and 49,and a vector of
⍝ 15 random numbers between 0 and 31.

?¨(10⍴50;15⍴32) ⍝ The same result as above. In this
⍝ case the each operator, denoted by
⍝ the dieresis, is applied to the
⍝ function ? and the vector (10ρ50;15ρ32).
⍝ The effect of the operator is to apply
⍝ the function to the contents of each
⍝ element of the vector.

⍳¨3 5 10 6 ⍝ A 4-element vector whose elements hold, in
⍝ order, the vecctor ⍳3, the vector ⍳5,
⍝ the vector ⍳10, and the vector ⍳6.

10?¨15 20 25 ⍝ A 3-element vector whose elements hold 10?15,
⍝ 10?20, and 10?25.

10 15?¨15 20 ⍝ A 2-element vector whose elements hold 10?15
⍝ and 15?20.

10 15?¨25 ⍝ A 2-element vector whose elements hold 10?25
⍝ and 15?25.

<@1 ⍳5 7 ⍝ A 5-element vector whose elements hold the
⍝ 7-element vectors that make up the rows of ⍳5 7.
```

3. Characteristics of Nested Arrays

The primitive functions and operators illustrated in this section are:
Shape: Monadic ρ
Indexing: [;⍳;]
Count and Choose: Monadic and Dyadic #
Type: Monadic ∨
Depth: Monadic ≡
Apply: Monadic operator denoted by ¨ that applies to function scalars

The purpose of this section, as in the Simple Arrays tutorial, is to illustrate the A+ primitives that deal with these most basic properties of arrays:

shape, e.g. the number of rows and columns of a matrix;

reshaping an array to a specified shape;

count, e.g. the number of rows of a matrix;

type, that is, whether the elements are characters, integers, etc.;

depth, that is, the levels of nesting;

selection of elements and subarrays.

Nesting adds a new aspect to arrays, namely, that elements of arrays do not have to be individual values, but can hold other arrays. Whether or not an array is nested has no effect on the basic functions for determining count and shape, for reshaping, and for selecting elements or subarrays, because the definitions of these functions do not depend on the contents of elements. For example, the vector

```
v←('abc';⍳5 2;`as_de;12 34 891)
```

has four elements and so its count is the scalar 4 and its shape is the one-element vector 1⍴4, just as the count of the four-element vector 3 14 21 ¯5 is the scalar 4 and its shape is the one-element vector 1⍴4:

```
#v
 4
⍴v
 4
```

Analogously, elements of nested arrays can be rearranged with Reshape and selected with Indexing and Choose, just like the elements of simple arrays:

```
a←2 2⍴v
```

For example, both a[0;1] and (0;1)#a equal the element at the intersection of the first row and second column, regardless of whether that element is nested or not. As in the simple array case, this element is a scalar; Indexing and Choose have not selected the array inside the element, namely ⍳5 2, but the element that holds that array:

```
a[0;1]
< 0 1 2 3 4  ⍝ The leading < indicates nesting.
  5 6 7 8 9
(0;1)#a
< 0 1 2 3 4
  5 6 7 8 9
⍴(0;1)#a
⍝ Only a new line occurs, indicating an empty result.
```

The Type primitive, denoted by monadic ∨, applies to nested arrays and returns `box as their type, as in:

```
Va
 ‘box
V(0;1)#a ⍝ The selected element is also boxed.
 ‘box
```

However, the situation is more complicated than this example indicates, and is discussed in the section that follows (3a1. Type).

The new primitive introduced in this section is Depth, denoted by monadic ≡. The purpose of this primitive is to serve an analogous role to the Shape primitive, measuring the level of nesting in an array instead of the lengths of axes. In particular, the depth of any simple array of type ‘int, ‘float, or ‘char, is 0. For example:

```
≡1 2 3 4
 0
≡'abcdef'
 0
≡a ⍝ At least one nested element 1 level down.
 1
```

As with the Type primitive, we will return to the Depth primitive in the next section (3a2. Depth).

3a. Nested Arrays, Function Arrays and Symbol Arrays

A symbol is a string of alphabetic characters, underscores (_), and dots (.) that is represented as a single scalar, just as an integer is a single scalar representing a string of digits. A symbol constant consists of a backquote (‘) followed by a string of characters. For example, just as 1 2.34 12e3 is a list of three numbers, ‘a.s ‘12 ‘w_3 is a list of three symbols.

A function scalar is a scalar array holding a function (not the name of the function, but the function itself). A constant symbol scalar consists of the symbols <{, followed by a function, followed by the symbol }. For example, <{+/} is a function scalar holding + Reduction. The sole purpose of the Apply operator is to evaluate function scalars. For example:

```
(<{+/})¨3 4 5 6 7
```

Symbol arrays and function arrays are arrays whose elements are all symbols or all function scalars, respectively.

3a1. Type

Function arrays, symbol arrays, and nested arrays are actually all variations of the same internal representation, and thereby can be combined freely to form other arrays. For example, the following expressions are valid:

```
‘abc,<⍳5 2
(<2 3⍴’abcdef’),<{+}
(<{×/}),‘times_reduction
```

On the other hand, the primitive function Type distinguishes between symbol, function, and nested arrays, as in the following examples:

```
∨‘abc
∨<{×/}
∨<’abc’
```

Consequently, one might say that arrays whose elements hold objects of more than one type are of mixed type, as in the case of the three examples above. Le’s see what the type function returns when applied to them:

```
∨‘abc,<⍳5 2
∨(2 3⍴’abcdef’;+)
∨(×/;‘times_reduction)
```

As you can see, there is no array type called ‘mixed. Instead, the type rule is as follows:

The type of a non-empty array is the type of the first element of its ravel.

For example, reverse each of the arrays in the above examples to get a new first element and apply the Type primitive again:

```
∨⌽‘abc,<⍳5 2
∨⌽(2 3⍴’abcdef’),<{+}
∨⌽(<{×/},‘times_reduction
```

There is a fourth player among the mixed arrays, the Null, which is denoted by (). The Null is the only empty vector to which any elements of type ‘box, ‘func, or ‘sym can be concatenated. It has its own type, which is ‘null. For example:

```
∨()
```

```
V(),‘abc
V(),<{ ,×}
V(),<⍳10
V(;⍳10)
```

All empty nested arrays are of type ‘null. For example:

```
a←⍳¨2 2⍴5 15 10 20  ⍝ A four-element vector of integer vectors.
⍴0 0/a
 0 2
V0 0/a
 ‘null
⍴2↓@1 a
 2 0
V2↓@1 a
 ‘null
```

Ex 1 What is the difference, if any, between ‘abc,<⍳3 4 and (‘abc;⍳3 4)?

3a2. Depth

In the previous section we saw that arrays of four different types can be combined to form other arrays. Depth reflects the level of nesting in arrays, and as such does not have the same value for scalars of these four types. For example:

```
≡‘abc
≡<{+.×}
≡<⍳10 4
≡<<⍳10 4
≡()
```

Ex 2 Here are the rules for the Depth primitive:

```
    the depth of a scalar of type other than ‘box is 0;
    the depth of an empty array of type other than ‘null is 0;
    every level of enclosure in a scalar increases the depth of the scalar
      by 1;
    the depth of an array is the maximum depth of its (scalar) elements.
```

Write a defined function that is equivalent to the Depth primitive.

Ex 3 We tend to use the terms symbol array for those arrays whose elements are all symbols, and function array for those arrays whose elements all hold function expressions, much in the same way as we use integer array, floating-point array, or character array. The latter three are easily defined, namely, as arrays of type ‘int, ‘float, ‘char, respectively. However, even if the type of an array is ‘sym, the array is not necessarily a symbol array; e.g., (‘a;3). And similarly for function arrays. Each of the following expressions attempts to test whether or not an array is a symbol array, or a function array. Which are successful, and for any that are not, why not?

```
a is a symbol array if (‘sym=a)∧0=≡a
x is a function array if ~0∊‘func=∨@0 x
xy is a symbol array if 0=0⊃do ⊤xy
z is a function array if ~0∊(<‘func)=∨¨z
```

3b. General Purpose Primitive Functions and Nested Arrays

Among the primitive functions discussed in the Simple Arrays tutorial are those that select and rearrange items and elements of their arguments. These functions generally apply to simple arrays of any type. It turns out that they apply to nested arrays as well, and in the expected manner; the definitions of these functions for selecting and rearranging items and elements do not depend in any way on the contents of those elements and items. We have already examples of this above, namely, Indexing and Choose.

For example, the Take primitive applies to nested right arguments as well as simple ones:

```
2↑‘xyz,(’abcd’;⍳4 2),<{+}
<  xyz
< abcd
```

4. Constructing Nested Elements and Extracting their Contents

The primitive functions illustrated in this section are:
Enclose: Monadic <
Disclose: Monadic >
Pick: Monadic ⊃

The basic buiding block for nested arrays is the Enclose primitive, denoted by monadic <. The Enclose primitive applies to any array and produces a scalar holding that array; the depth of the scalar result is 1 plus the depth of the argument, and the type of the result is always ‘box.

```
<'abcde'
< abcde
ρ<'abcde'
⍝ Only a new line occurs, indicating an empty result.
≡'abcde'
 0
≡<'abcde'
 1
≡<<'abcde'
 2
```

The Disclose primitive, denoted by monadic >, is a left inverse of Enclose, in that the Disclose of the Enclose of any array equals that array, unless that array was a function: the depth of a function is -1, and the minimum depth of the result of Disclose is 0.

```
><'abcde'
 abcde
'abcde'≡><'abcde'
 1
```

Disclose is also permissive, in that applies to scalars of depth 0, i.e. scalars that are not a result of Enclose.

```
>3
 3
3=>3
 1
```

Disclose also applies to arrays with more than one element, as long the all the disclosed elements have the same shape:

```
>(1 2 3 4; 60 70 80 90)
  1  2  3  4
 60 70 80 90
```

In its simplest form, the Pick primitive, denoted by dyadic ⊃, is Choose followed by Disclose:

```
1⊃('xcty';`xy_w;⍳5 7)
 `xy_w
>1#('xcty';`xy_w;⍳5 7)
 `xy_w
`xy_w≡1⊃('xcty';`xy_w;⍳5 7)
 1
```

```
2⊃('abc';⍳5 2;`as_de;12 34 891)
 `as_de
>2#('abc';⍳5 2;`as_de;12 34 891)
 `as_de
```

The left argument to Pick can also be a path vector that reaches as deep into the right argument as its length. For example:

```
0 1⊃(('abcd';3 4 1);`xyz)
 3 4 1
```

Ex 4 In the above examples the left arguments to Pick are composed of index expressions for the various levels of the corresponding right arguments. Note the order of these index expressions in the left arguments. Namely, if the left argument x is a vector with more than one element, and is a valid path vector for the right argument w, then x⊃s equals (1↓x)⊃(1↑x)⊃w.

    Test this relation for the appropriate examples above;

What are the conditions on x and the scalar n so that x⊃w equals (n↓x)⊃(n↑x)⊃w.

4a. Slot-Filler Arrays

Slot-filler arrays are two-element nested vectors that satisfy certain requirements and that, in return, are treated specially by the Pick primitive and screen display functions. Specifically, an array s is a slot-filler array if the following conditions are met:

    s is a two-element vector;

both 0⊃s and 1⊃s are vectors or both are scalars, and they have the same number of elements;

    all the elements of 0⊃s are symbols;
    the depth of 1⊃s is at least 1.

For example, the following are slot-filler arrays:

```
(`a`b`c;(10;20;30))
(`w`xx`yyy`zzzz;(;⍳5 3;'abcdef';(`astd;3 4 5 1)))
(`qw;<10?20)
```

The following are not:

```
(`a`b;(10;30;30))
(`x`y`zz;2 3 4)
```

Ex 5 Define a monadic function that returns 1 if its argument is a slot-filler array and 0 otherwise. Test your function on a variety of examples, and compare the results with those of the system function _issf.

From the viewpoint of the Pick primitive, slot-filler arrays are key-value pairs. That is, if s denotes a slot-filler array, a symbol in 0⊃s can be used as the left argument to Pick, with s the right argument, and the result is the contents of the corrseponding element of 1⊃s. For example:

```
`a⊃(`a`b`c;(10;20;30))
 10
`b⊃(`a`b`c;(10;20;30))
 20
```

Ex 6 If s is a slot-filler array and x is a symbol that appears in 0⊃s, then x⊃s equals ((0⊃s)⍳x)⊃1⊃s. Is this definition of x⊃s correct? If not, correct it.

A nested slot-filler array is one whose values in the key-value pairs may be also slot-filler arrays themselves, or even nested slot-filler arrays. For example, the following is a nested slort-filler array:

```
s←(`a`b`c;(10;(`x`y;(100;200));(`e`f;((`g`h;(1000;2000));'A+'))))
```

when the right argument to Pick is a nested slot-filler array, the right argument can be a vector of symbols reaching into the right argument to extract a value. For example, evaluate each of the following:

```
`a⊃s
`b`x⊃s
`b⊃s ⍝ There is no need to Pick all the way to the bottom.
`c`f⊃s
`c`e`g⊃s
```

The symbol scalar or vector on the left of Pick is called a path to the slot-filler array on the right.

Ex 7 Repeat Ex 4 for nested slot-fillers. Note the order of the symbols in the left argument to Pick in the above examples. Namely, if the left argument v is a vector with more than one element, and is a valid path vector to the nested slot-filler array s, then v⊃s equals (1↓v)⊃(1↑v)⊃s.

Test this relation for the appropriate examples above;

What are the conditions on v and the scalar n so that v⊃s equals (n↓v)⊃(n↑v)⊃s.

5. Primitive Functions and Operators Specifically for Nested Arrays

Some of the primitive functions discussed in the Simple Arrays tutorial apply to nested arrays, while others are restricted to simple arrays - in particular, those that are further restricted to numeric arrays. There are also primitives that are specific to nested arrays, in that their arguments, or results, or both, must be nested arrays. It is these primitives that are discussed in this section.

5a. Partition, Partition Count, and the Each Operator

The primitives illustrated in this section are useful for partitioning arrays into sections. For example, suppose we want to break the following sentence into words:

```
s←"Partition Count is a dyadic function, while Partition is monadic."
```

We begin by identifying the word breaks, namely the blanks:

```
b←s=' '
b
 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0
 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0
```

The Partition Count primitive applies to b and returns a vector whose length is the number of 1↓s in b, i.e. +/b, and whose ith item is 1 plus the length of the sequence of contiguous 0's following the ith 1. The argument to Partition Count must start with a non-zero element. Since the sentence s does not have a leading blank, b does not have a leading 1, so we'll prefix a 1 to b:

```
⊂1,b
 10 6 3 2 7 10 6 10 3 9
```

Be sure to compare this result to the lengths of sequences of 0's in

b. The Partition function will break the sentence s up into parts (words, hopefully) based on the last result.

```
  (⊂1,b)⊂s
< Partition
< Count
< is
< a
< dyadic
< function,
< while
< Partition
< is
< monadic.
```

As you can see, we have a nested vector of character vectors, which are (almost) words. Nested arrays are very useful here, because the character vectors are of different lengths. Let's check the lengths of words by applying the Count primitive to each element of the result (by way of the Each operator):

```
w←(⊂1,b)⊂s
>#¨w
 10 6 3 2 7 10 6 10 3 8
```

Note that these values agree with the elements in ⊂1,b (there is no trailing blank on the last item). Each character vector has at least one unwanted character. For example, the first character vector holds the word "Partition", but the vector has 10 characters and the word has only 9. We can check that the extraneous characters are not on the front of the words by looking at the first character of each one:

```
1↑¨w
< P
< C
< i
< a
< d
< f
< w
< P
< i
< m
```

So, the extraneous characters are on the ends of the words. We will continue this example in the exercises.

Ex 8 Use the Drop primitive and the Each operator to remove one character from the end of each word. Which of the resulting character vectors still contain extraneous characters? Suggest ways to get rid of all extraneous characters at once.

Ex 9 Insert an extra space between two words in the sentence s, and call the new sentence s0. For example:

```
s0←"Partition Count  is a dyadic function, while Partition is monadic."
```

Partition s0:

```
w0←(⊂1,s0=' ')⊂s0
```

Examine w0, and explain how the extra blank in s is manifested in w0. How would you remove it from the w0?

Ex 10 Partition the vector s0 of Ex 9 as follows:

```
w1←(⊂1,s0∊' ,.')⊂s0
```

Examine w1, and explain how the punctuation characters, as well as the extra blank, are manifested in w1. How would you remove these characters from w1?

Ex 11 Suppose the leading character is a blank, i.e. form

```
s1←' ',s
```

Describe the result of

```
w2←(⊂1,s1∊' ,.')⊂s1
```

Ex 12 How would you define a function to partition a sentence into words, when the punctuation characters can be commas, periods, and semicolons, and when there can be multiple blanks between words, and zero or mare blanks at the beginning and end of the sentence?

5b. Raze and Rake

The primitives Raze and Rake apply to nested arrays and bring all elements up level 0 and 1, respectively.