```
⍝ APL Array Notation - Phil Last - phil.last@4xtra.com

      This is the session log from the live presentation given via webcast as a part of
      Dyalog'15 - Sicily on 2015-09-08 and annotated with these offset insertions.

⍝ Necessarily a precis - omissions - limitations - anomalies

      a short talk with a simple message but one that throws up a lot of questions - at
      least it did in the author's mind - unit dimensions, empty arrays & dictionaries
      among them.

⍝ Model - expr - value - fmt - eval - edit

      Obviously APL will not anticipate the changes I'd like to see so I've written a
      model to show it.

⍝ Here are some expressions

      expr⊢A←0 1 2 3                           ⍝ here is a vector
(0 1 2 3)

      expr⊢B←'this string'                     ⍝ and a string
'this string'

      expr⊢C←(4 5 6)(7 8 9)                    ⍝ a nested vector
((4 5 6)(7 8 9))

      expr⊢D←'three' 'text' 'strings'          ⍝ list of strings
('three' 'text' 'strings')

      expr⊢E←0 '1' 2 '3' 4 '5' 6               ⍝ mixed vector
(0 '1' 2 '3' 4 '5' 6)

      expr⊢F←'0' 1 '2' 3 '4' 5 '6'             ⍝ and another
('0' 1 '2' 3 '4' 5 '6')

      ]disp ⍉¨(expr¨,⊢,disp¨)⍪A B C D E F  ⍝ here they all are
┌→──────────────────────────┬───────────────────────┬─────────────────────────────┐
↓(0 1 2 3)                  │0 1 2 3                │┌→──────┐                    │
│                           │                       ││0 1 2 3│                    │
│                           │                       │└~──────┘                    ↓
├──────────────────────────→┼──────────────────────→┼────────────────────────────→┤
│'this string'              │this string            │┌→──────────┐                │
│                           │                       ││this string│                │
│                           │                       │└───────────┘                ↓
├──────────────────────────→┼──────────────────────→┼────────────────────────────→┤
│((4 5 6)(7 8 9))           │ 4 5 6  7 8 9          │┌→─────────────────┐         │
│                           │                       ││ ┌→────┐ ┌→────┐  │         │
│                           │                       ││ │4 5 6│ │7 8 9│  │         │
│                           │                       ││ └~────┘ └~────┘  │         │
│                           │                       │└∊─────────────────┘         ↓
├──────────────────────────→┼──────────────────────→┼────────────────────────────→┤
│('three' 'text' 'strings') │ three  text  strings  │┌→──────────────────────────┐│
│                           │                       ││ ┌→────┐ ┌→───┐ ┌→──────┐ ││
│                           │                       ││ │three│ │text│ │strings│ ││
│                           │                       ││ └─────┘ └────┘ └───────┘ ││
│                           │                       │└∊──────────────────────────┘↓
├──────────────────────────→┼──────────────────────→┼────────────────────────────→┤
│(0 '1' 2 '3' 4 '5' 6)      │0 1 2 3 4 5 6          │┌→────────────┐              │
│                           │                       ││0 1 2 3 4 5 6│              │
│                           │                       │└+────────────┘              ↓
├──────────────────────────→┼──────────────────────→┼────────────────────────────→┤
│('0' 1 '2' 3 '4' 5 '6')    │0 1 2 3 4 5 6          │┌→────────────┐              │
```

```
│                            │                       ││0 1 2 3 4 5 6│             │
│                            │                       │└+────────────┘             ↓
└───────────────────────────→└──────────────────────→└───────────────────────────→┘

⍝ expression - default - DISPLAY
⍝ explicitness - but ambiguity in mixed and multi-dimensional arrays

      I'm a bit disparaging of both the default and the DISPLAY formats for their
      ambiguity but three decades ago I wrote an array editor using DISPLAY and ⎕SM,
      converting the simple content to input fields - it did the job.

⍝ expressions never ambiguous -
⍝ but have limitations - singletons - multi-dimensions not in notation alone

      vector notation is incapable in itself of denoting a one item vector, string or
      list, it can an empty string but not an empty vector or list and certainly not a
      multi-dimensional array

⍝ even in a simple example we have to work harder -

      - to understand what is being created

      (2 3⍴0 1 2 3 4 5)                     ⍝ multi-dim
0 1 2
3 4 5

   ⍝  ((0 1 2)(3 4 5))                      ⍝ vector

      whereas we don't even have to execute vector notation to know the result

⍝ How to express multi-dimensions in notation?
⍝ Why do we need to?
⍝ Static arrays - storage - generation - maintenance - non-simple -

      all systems have arrays that are static and used throughout the session. Many
      developers consider it anathema to store arrays with code and of course storing
      code as scripts precludes it. Assignment within the script or in a function
      becomes necessary as well as desirable. In any case maintenance makes the
      generation of the array at start-up seem a more parsimonious proposition as we
      store source rather than data.

      ]display ⊢ET←(⊂1 11 59 64 69 81)⌷DRC.ErrorTable
┌→────────────────────────────────────────────────────────────────────┐
↓                                                                     │
│ 100  ┌→──────┐          ┌⊖┐                                         │
│      │TIMEOUT│          │ │                                         │
│      └───────┘          └─┘                                         │
│      ┌→──────────────┐  ┌⊖┐                                         │
│ 1009 │ERR_NAME_IN_USE│  │ │                                         │
│      └───────────────┘  └─┘                                         │
│      ┌→──────────┐      ┌→───────────────────────┐                  │
│ 1108 │ERR_TIMEOUT│      │/* Request timed out */ │                  │
│      └───────────┘      └────────────────────────┘                  │
│      ┌→─────────────────┐ ┌→─────────────────────────────────────────┐│
│ 1113 │ERR_RECV_CMDANSWER│ │/* Could not receive CMD answer from host */││
│      └──────────────────┘ └─────────────────────────────────────────┘│
│      ┌→────────────┐    ┌→──────────────────────────────────────┐   │
│ 1118 │ERR_SEND_NACK│    │/* Could not send nack packet to client */│ │
│      └─────────────┘    └───────────────────────────────────────┘   │
│      ┌→──────────────┐  ┌→────────────────────────────────────┐     │
│ 1130 │ERR_WRONG_MAGIC│  │/* Package with wrong magic received */│   │
│      └───────────────┘  └─────────────────────────────────────┘     │
└∊────────────────────────────────────────────────────────────────────┘
```

error table from DRC - must have been a real chore for Bjørn to maintain during development of the system

```
⍝ Alternatives -

      ⊢M←demo.example0                     ⍝ here is a simple array
first row of matrix
second row of matrix
third row of matrix

      ⎕TF 'M' ⍝ APL2 solution - executable expression with reshape
SYNTAX ERROR
      ⎕TF  'M' ⍝ APL2 solution - executable expression with reshape
         ^

      'M←3 20 ⍴''first row of matrix second row of matrixthird row of matrix '''
M←3 20 ⍴'first row of matrix second row of matrixthird row of matrix '
```

⎕TF works in APL2 but the result is neither editable nor human readable

```
      ⊢M←3 20 ⍴'first row of matrix second row of matrixthird row of matrix '
first row of matrix
second row of matrix
third row of matrix

      ⎕cr'#.demo.create0'                  ⍝ repeated catenation
 create0←{
     r←⍬
     r,←⊂'first row of matrix'
     r,←⊂'second row of matrix'
     r,←⊂'third row of matrix'
     ↑r
⍝ repeated catenation
 }
```

a tried and tested technique wherein required rows are marked up by catenation to a result variable

```
      #.demo.create0 0
first row of matrix
second row of matrix
third row of matrix

      ⎕cr'#.demo.create1'                  ⍝ extraction of comments
 create1←{
     {⍵/⍨' '∨.≠⍵}0 2↓{⍵⌿⍨>/'⍝-'⍷⍵}⎕CR⊃⎕SI
⍝-  first row of matrix
⍝-  second row of matrix
⍝-  third row of matrix
⍝ comment extraction
 }
```

another technique with a long pedigree wherein required rows are marked up with comment symbols and extracted from the ⎕CR. The generation of ⎕CR in "create1" above is actually more expensive than the entire call to "create0" 'though not something over which to lose sleep.

```
      #.demo.create1 0
first row of matrix
second row of matrix
third row of matrix
```

```
⍝ ATTACHED_TEXT - ADOC - ScriptFollows - forum discussion
```

examples of comment extraction - ATTACHED_TEXT: the author's early attempt from the bottom of the calling function, ADOC: Kai's documentation facility that extracts text and mark-up from the script to create various document types and ScriptFollows: Brian's utility for generating javascript where it's needed.

```
⍝ Desirability of expressive code
```

- is shown by each of the above techniques with varying degrees of success

```
⍝ I want to code a three row matrix as three lines of code.
⍝ Implies 3-line expression without a function call!!!!
```

just as with vector notation the definition of a multi-dimensional array should look as much like the final array as possible - with absolutely minimal mark-up

```
⍝ Let's look at a dfn -

      ⎕cr'demo.dfn0'                             ⍝ one line
 dfn0←{
     0 1({z←,⊂'first line' ⋄ z,←⊂'second line' ⋄ ↑z,⊂'third line'})6 7
⍝
 }
```

wherein an embedded dfn is parenthesised

```
      demo.dfn0 0
first line
second line
third line

      ⎕cr'demo.dfn1'                             ⍝ multi-line
 dfn1←{
     0 1({                   ⍝ ←      syntax
         z←,⊂'first line'    ⍝     colouring
         z,←⊂'second line'   ⍝     indicates
         ↑z,⊂'third line'    ⍝    unbalanced
     })6 7                   ⍝ ← parentheses
⍝
 }
```

wherein the diamonds are replaced with line-ends

```
      demo.dfn1 0
first line
second line
third line
```

it still works because the line-ends are embedded in the dfn

```
      ⎕cr'demo.dfn2'                             ⍝ broken line
 dfn2←{
     0 1(                    ⍝ ←   line-end
     {                       ⍝   is outside
         z←,⊂'first line'    ⍝       braces
         z,←⊂'second line'   ⍝      causing
         ↑z,⊂'third line'    ⍝       syntax
     })6 7                   ⍝        error
⍝
```

```
 }
```

but not when a line-end falls outside the braces

```
      demo.dfn2 0
SYNTAX ERROR
dfn2[2] 0 1(                  ⍝ ←   line-end
       ^
      →

      ⎕cr'#.demo.create2'                     ⍝ array notation
 create2←{
     ['first row of matrix'
     'second row of matrix'
     'third row of matrix']
⍝ array notation
 }
```

here we borrow dfns' ability to embed line-ends but in brackets [] instead of braces {} and we want APL to understand that the resultant unbalanced brackets - or rather the line-ends between them - can only indicate the start of a new cell in a multi-dimensional array - unlike those between braces which mark the start of a new statement

```
⍝ Doesn't have to be brackets - many reasons why not
⍝    [ ⋄ ]    ( ⋄ )    [[ ⋄ ]]    ([ ⋄ ])    ⍃ ⋄ ⍄
⍝ Problem with digraphs - human parsing - unicode - many bracketing pairs.
```

I have used brackets hereinafter 'though any matching pair apart from braces would do.

```
      #.demo.create2 0
SYNTAX ERROR
create2[1] ['first row of matrix'
          ^
      →
```

of course it doesn't work because APL doesn't actually know yet!!

```
      ⎕cr'#.demo.create3'                     ⍝ model implementation
 create3←{{
         ['first row of matrix'
         'second row of matrix'
         'third row of matrix']
     }##.value''
⍝ array notation
 }
```

here we demonstrate the possibility by embedding the array notation in an internal dfn and passing it as operand to an operator, a part of the model, that extracts the data

```
      #.demo.create3 0
first row of matrix
second row of matrix
third row of matrix

      ⎕cr'demo.threeDim'                      ⍝ into three dimensions
 threeDim←{{
         [
         ['first' 'plane' 'first' 'row'
         'first' 'plane' 'second' 'row']
```

```
          ['second' 'plane' 'first' 'row'
          'second' 'plane' 'second' 'row']
          ]
      }##.value''
 }
```

and creating three dimensions merely requires another pair of delimiters

```
      ⊢M←demo.threeDim 0                      ⍝ default display
 first   plane  first   row
 first   plane  second  row

 second  plane  first   row
 second  plane  second  row

      ]display M                              ⍝ DISPLAY
┌┌→────────────────────────────────────┐
↓↓ ┌→────┐  ┌→────┐ ┌→────┐  ┌→──┐   │
││ │first│  │plane│ │first│  │row│   │
││ └─────┘  └─────┘ └─────┘  └───┘   │
││ ┌→────┐  ┌→────┐ ┌→─────┐ ┌→──┐   │
││ │first│  │plane│ │second│ │row│   │
││ └─────┘  └─────┘ └──────┘ └───┘   │
││                                    │
││ ┌→─────┐ ┌→────┐ ┌→────┐  ┌→──┐   │
││ │second│ │plane│ │first│  │row│   │
││ └──────┘ └─────┘ └─────┘  └───┘   │
││ ┌→─────┐ ┌→────┐ ┌→─────┐ ┌→──┐   │
││ │second│ │plane│ │second│ │row│   │
││ └──────┘ └─────┘ └──────┘ └───┘   │
└└∊────────────────────────────────────┘
```

the model's "fmt" function could be a new calling sequence of format (⍕)
returning a multi-line array expression ready to embed in a function

```
      fmt 2 3⍴⍳6                              ⍝ so if this
[0 1 2
3 4 5]

      2 3⍴⍳6                                  ⍝ means this
0 1 2
3 4 5
```

- as could the "expr" function returning a diamond delimited version of the same

```
      expr 2 3⍴⍳6                             ⍝ then so should this
[0 1 2⋄3 4 5]
```

the "eval" function takes an "expr" or "fmt" result and evaluates it returning
the represented array. No analogous primitive would be needed as the "expr" and
"fmt" results would be notation - just code - and evaluated as a matter of
course.

```
      eval ⍞                                  ⍝ and so it does
[0 1 2⋄3 4 5]
0 1 2
3 4 5

      fmt M                                   ⍝ and this
[['first' 'plane' 'first' 'row'
'first' 'plane' 'second' 'row']
['second' 'plane' 'first' 'row'
'second' 'plane' 'second' 'row']]

      expr M                               ⍝ and this are also equivalent
[['first' 'plane' 'first' 'row'⋄'first' 'plane' 'second' 'row']⋄['second'
'plane' 'first' 'row'⋄'second' 'plane' 'second' 'row']]

      eval ⍞
[['first' 'plane' 'first' 'row'⋄'first' 'plane' 'second' 'row']⋄['second'
'plane' 'first' 'row'⋄'second' 'plane' 'second' 'row']]
 first   plane  first   row
 first   plane  second  row

 second  plane  first   row
 second  plane  second  row

      ⎕cr'edit'                            ⍝ editor
 edit←{
     r←#.fmt ⍵
     z←⎕ED'r'
     #.eval r
⍝ not quite David Liebtag
 }
```

the "edit" function creates a multi-line expression, allows the user to edit it and evaluates the result. Something the function editor could do rather easily.

(David Liebtag created a professional version of an array editor based upon the DISPLAY format but using somewhat more advanced techniques than ⎕SM!)

```
      N←edit M
```

wherein this

```
[['first' 'plane' 'first' 'row'
'first' 'plane' 'second' 'row']
['second' 'plane' 'first' 'row'
'second' 'plane' 'second' 'row']]
```

is edited to become this

```
[

['first' 'plane' 'first' 'row'
'first' 'plane' 'second' 'row']

['second' 'plane' 'first' 'row'
'second' 'plane' 'second' 'row']

['another' 'plane']

]
```

which is evaluated to become this

```
      ]display N
┌┌→─────────────────────────────────────────┐
↓↓ ┌→────┐    ┌→────┐    ┌→────┐    ┌→──┐     │
││ │first│    │plane│    │first│    │row│     │
││ └─────┘    └─────┘    └─────┘    └───┘     │
││ ┌→────┐    ┌→────┐    ┌→─────┐   ┌→──┐     │
││ │first│    │plane│    │second│   │row│     │
││ └─────┘    └─────┘    └──────┘   └───┘     │
││                                            │
││ ┌→─────┐   ┌→────┐    ┌→────┐    ┌→──┐     │
││ │second│   │plane│    │first│    │row│     │
││ └──────┘ └─────┘ └──────┘ └──┘           │
││                                          │
││ ┌→─────┐ ┌→────┐ ┌→─────┐ ┌→─┐           │
││ │second│ │plane│ │second│ │row│          │
││ └──────┘ └─────┘ └──────┘ └───┘          │
││                                          │
││ ┌→──────┐ ┌→────┐ ┌→──────┐ ┌→──────┐    │
││ │another│ │plane│ │       │ │       │    │
││ └───────┘ └─────┘ └───────┘ └───────┘    │
││ ┌→──────┐ ┌→──────┐ ┌→──────┐ ┌→──────┐  │
││ │       │ │       │ │       │ │       │  │
││ └───────┘ └───────┘ └───────┘ └───────┘  │
│└∊─────────────────────────────────────────┘
```

somehow I managed to substitute M (and N) above for the intended ET seen about a quarter of the way into the talk. Here is it's fmt

```
      disp fmt ET
┌→──────────────────────────────────────────────────────────────────────┐
│[100 'TIMEOUT' ''                                                      │
│1009 'ERR_NAME_IN_USE' ''                                              │
│1108 'ERR_TIMEOUT' '/* Request timed out */'                           │
│1113 'ERR_RECV_CMDANSWER' '/* Could not receive CMD answer from host */'│
│1118 'ERR_SEND_NACK' '/* Could not send nack packet to client */'      │
│1130 'ERR_WRONG_MAGIC' '/* Package with wrong magic received */']      │
└───────────────────────────────────────────────────────────────────────┘
```

eminently editable

```
⍝ That's all folks!
```

There were two questions from the floor.

Bob Bernecky asked how unprintable characters would be handled as they can occur fairly frequently in data.

The model utilises a technique, similar 'though simplified compared with that in JSON, wherein any character can be represented as an escape character followed immediately by four hex digits. The escape character, when required as a literal, must be represented as itself followed by its hex equivalent. Of course they can only ever appear in quotes as they are text.

Jay Foad asked how nested arrays could be handled within array notation.

My answer assumed the generalisation of the question in that expressions for nested arrays or anything else are just code and given that any reasonable language permits replacement rules and APL has (almost) no limitations, if we can write

```
      [0 1 2
       3 4 5]
```

we can equally write

```
      [f w
       f.g/x y z]
```

whose rank will be one more than the greater of the ranks of the two expressions.

What distinguishes that as "array" notation is the presence of the bracketed line-ends.

The rest of this document attempts to answer the questions, limitations and anomalies that I met in developing the model and defining the script for the presentation.

"any matching pair apart from braces would do"

Braces are precluded as delimiters for array notation for the simple reason that line-ends are already permitted directly between them as statement separators in dfns. Given that no other bracketing pair has that distinction then their inclusion would engender a SYNTAX ERROR and their permitting would be a conforming extension. Of currently implemented delimiters this leaves parentheses () and brackets []. Parentheses have been used elsewhere in a different context: APL#, an experimental interpreter from Dyalog, where a multi-line statement was a kind of semi-dfn wherein assignments were only localised in the surrounding scope. I used brackets in the model and presentation. No conflict arises with indexing and axis because neither permits directly embedded line-ends. Were we to allow that [0 1 2 3] represents a one row matrix (without line-ends) then this would conflict with index or axis if a function or array were adjacent on the left. This could be overcome by parentheses:

```
array[0 1 2 3]           ⍝ indexed array
array([0 1 2 3])         ⍝ two-vector
function[0 1 2 3]array   ⍝ function with axis
function([0 1 2 3])array ⍝ function applied to two-vector
```

Alternatively a one row array could be represented as [⋄0 1 2 3] where the diamond forces the array but there is only one expression.

So far we haven't mentioned objects (or dictionaries or spaces).

Double brackets [[ ]] have been proposed as a notation for namespaces, originally in APL# but perhaps as an extension to DyalogAPL. Assignments between the brackets, separated by line-ends or diamonds, would define the members of the space.

```
obj←[[str←'this' ⋄ vec←3 4 5 ⋄ sub←[[z←0]] ]]
```

Note the visual, 'though perhaps not syntactic, necessity of the white-space between the closing bracket pairs where the final member is a space. Only the presumed definition of an empty space as [[]] prevents the redundancy of the digraphs.

> Just as the distinction between the empty object {} and empty list [] are the only reason that different delimiters are required for objects and arrays in JSON. They could as easily have been [:] and [].

The following would work as well for APL in the avoidance of both redundancy and ambiguity.

```
empty space:  [←]
one-member:   [n←v]
multi-member: [n0←v0 ⋄ n1←v1 ⋄ ...]
empty array:  [⋄]
one-row:      [⋄v]
multi-row:    [a ⋄ b ⋄ ...]
              where the brackets could be any pair of delimiters and the diamonds,
              which could be equally line-ends, would be redundant in the empty
              and one-row cases if the delimiter were other than brackets.
```

A fine point. The presence or absence of "←" at the same depth of punctuation, as the distinguishing mark of spaces, presupposes a certain degree of simplicity of style. We can always confuse the issue by both assigning and returning a value as in

```
[ ⊢a←0 ⋄ ,b←3 ] ⍝ array or space?
```

The author has not decided this question. There is a fine distinction in dfns wherein the result of a function is the result of the first expression whose result is not absorbed by assignment. But arrays do not contain members. Perhaps the assignment arrow should be prohibited within array definitions unless embedded within spaces that are items of the array or within dfns that are called to generate items. A related point is whether expressions that do not assign names to values should be permitted in space definitions.

Another open question regards arrays with unit dimensions.

As mentioned in the presentation, vector notation alone cannot define a one item vector. In array notation what can [0 ⋄ 1 ⋄ 2] mean? The values of all individual expressions are scalars. If the rank of an array in array notation is determined as one more than the greatest rank of any of its cells then [0 ⋄ 1 ⋄ 2] must be the vector (0 1 2). This seems a waste of something potentially useful. But if we add the perfectly reasonable condition that all cells have rank of at least one, we are after all trying to define multi-dimensional arrays, then [0 ⋄ 1 ⋄ 2] is a three by one matrix. Of course the use of array notation will not preclude the use of functions any more than vector notation does so in the few cases where a limitation of vector notation affects the utility of array notation we can always force the issue with [,0 ⋄ 1 ⋄ 2].

My hope is that Dyalog and as many as possible of other APL vendors will consider these proposals and questions and perhaps come to some kind of agreement to include at least the basic concepts into their products.

Phil Last, <phil.last@4xtra.com> Wiltshire, Wednesday 9 Sept 2015