```
--------------------------------------------------------------------------------------------------
Proposal       Embedding data in code.
--------------------------------------------------------------------------------------------------
Justification
            ⊛  A recent thread in the Dyalog forums regarding JSON revived my
               interest in devising a native notation for embedding arrays
               in APL functions without repeated catenation and assignment.
            ⊛  The current output is formatted from what I refer to as an attached-text to a
               function. The function in question calls another utility that returns a selection
               from the ⎕CR of its caller. I think I remember basing this or at least attempting
               to justify it on a vague memory of a "Data" facility in the "Basic" language.
            ⊛  Since the advent of direct functions in Dyalog APL wherein we can embed functions
               between left and right braces "{}" within another function I have felt that
               it should be possible to do a similar thing to embed multi-dimensional and/or
               nested data as a part of APL over and above the familiar vector notation.
            ⊛  There follow two related proposals for a data notation, either of which could be
               embedded into a traditional function or operator or a script while the second,
               not being a "control structure", would also be suitable in a direct function.
            ⊛  The great advantage of both notations to the programmer is that the format of
               a constant numeric array or the quotation of a character array can be embedded
               readily into a function with little intervening markup, is easily both
               readable and editable in that context and becomes effectively self redefining.
            ⊛  In what follows every use of diamond "⋄" can equally
               be a newline (U+000D) character and vice versa.
--------------------------------------------------------------------------------------------------
:Array         Control Structure
            ⊛  This protocol is for programmers who like lots
               visual clues as to what the code is doing.
            ⊛  Definition:
               ,------------,-------------------------------------------------------------------,
               |            |                                  ,-<--------------<-,             |
               | Array      |  ":Array"-,- name -,- ":Mix" -,-'- "⋄" Expression -'- "⋄ :EndArray" |
               |            |           '->----->-'->------>-'                                   |
               |            |                                                                    |
               | Expression | -,- data-valued expression -,-                                     |
               |            |  '---------- Array ---------'                                      |
               '------------'-------------------------------------------------------------------'
               Being a control structure this construct is not assignable but its
               name, that is a part of the definition, could be localised in a header.
--------------------------------------------------------------------------------------------------
          [0]  White space and contiguous line-ends are ignored.
          [1]  A single expression adds no rank or depth:
```

```
,--------------------------------,----,------------,
| :Array name ◇ expr0 ◇ :EndArray | ↔ | name←expr0 |
'--------------------------------'----'------------'
```

[2] Each subsequent expression adds another item to a vector whose length will be the number of expressions and whose (absolute) depth will be 1 more than the maximum absolute depth of all items:

```
,--------------------------------------,----,---------------------,
| :Array name ◇ expr1 ◇ expr2 ◇ :EndArray | ↔ | name←(expr1)(expr2) |
'--------------------------------------'----'---------------------'
```

[3] With the :Mix keyword each subsequent expression adds another Cell to a resultant array whose rank is determined as one more than the maximum rank of all Cells.

```
,-----------------------------------------,----,-----------------------,
| :Array name :Mix ◇ expr1 ◇ expr2 ◇ :EndArray | ↔ | name←mix(expr1)(expr2) |
'-----------------------------------------'----'-----------------------'
```

where mix is equivalent to the final step of the operation of the Rank and Key operators where items having potentially different ranks are assembled.

[4] Array definitions are recursive; sub-arrays need not be named:

```
,-------------,----,-----------------------------------------,
| :Array name | ↔ | name←expr3((expr4)(expr5))(expr6)(expr7) |
|   expr3     |    |                                         |
|   :Array    |    |                                         |
|      expr4  |    |                                         |
|      expr5  |    |                                         |
|   :EndArray |    |                                         |
|   expr6     |    |                                         |
|   expr7     |    |                                         |
| :EndArray   |    |                                         |
'-------------'----'-----------------------------------------'
```

Possible extensions:

⊛ Within an :Array statement the synonyms ":Cell" and ":EndCell" could be used for cosmetic reasons only:

```
,-------------,
| :Array name |
|   expr3     |
|   :Cell     |
|      expr4  |
|      expr5  |
|   :EndCell  |
|   expr6     |
|   expr7     |
| :EndArray   |
'-------------'
```

- Entire cells' existence could be conditional:

```
,---------------,
| :Array name   |
|   expr3       |
|   :If cond    |
|     :Array    |
|        expr4  |
|        expr5  |
|     :EndArray |
|   :EndIf      |
|   expr6       |
|   expr7       |
| :EndArray     |
'---------------'
```

- :Return could be used for early truncation.

[Array] notation

- This construct has a more functional (and limited) nature but unlike the control structure it is also suited to be embedded in a direct function.
- Definition:

```
,------------,--------------------------------,
| Array      |      ,-<--- "⋄" --<-,          |
|            | "[" -'- Expression -'- "]"     |
|            |                                |
| Expression | -,- data-valued expression -,- |
|            |  '---------- Array ---------'  |
'------------'--------------------------------'
```

We utilise and reverse the fact that APL expressions as currently defined never start with "[". The other syntax change being that brackets "[]" are balanced around diamonds and new-lines rather than between them. Being an expression rather than a contol structure this can be directly assigned. Indeed it must be if it is not used within the same expression or returned as the result of a direct function.

- Points [0], [1] and [4] above apply equally to this notation.
- Only one of points [2] and [3] would apply, as the ":Mix" keyword is necessarily unavailable, unless some other provision were made to decide the issue on a case-by-case basis.
- If used without assignment this would require non-redundant parentheses to avoid SYNTAX ERRORS due to index and axis uses of brackets.
- For function application:

```
,-------------------------,----,-----------------------------------------,
| name←⍤1[expr3 ⋄ expr4]  |    | SYNTAX ERROR Invalid axis specification |
```

name←⍎([expr3 ⋄ expr4])	↔	name←⍎(expr3)(expr4)

⊛ For array justaposition:

name←X[expr5 ⋄ expr6]		SYNTAX ERROR Invalid index specification
name←X([expr5 ⋄ expr6])	↔	name←X((expr5)(expr6))

---

example:

```
schema←[
        [
    'Idtable'    'Name'
    'Name'       'Value'
    'Char(20)'   'Blob'
    ] ⋄ [
    'Codetable'  'Itemname'
    'Itemname'   'APLType'    'APLVersion' 'Value'   'Comment'  'Size'  'Upload'
    'CharW(80)'  'Dec(1)'     'Char(20)'   'Blob'    'Blob'     'Int32' 'Char(40)'
    ] ⋄ [
    'Fnametable' 'Filename'
    'Filename'   'Size'       'Hash'       'Comment' 'Upload'
    'Char(256)'  'Int32'      'Char(32)'   'Blob'    'Char(40)'
    ] ⋄ [
    'Fdatatable' '[Filename]'
    'Filename'   'Value'
    'Fnametable' 'Blob'
    ]
  ]
```

The result of the above is a 4-list of 3-lists each of which is a list
of strings the whole being the schema for a code repository database.
